姚新颜　编著

C/C++

深层探索

人民邮电出版社

图书在版编目(CIP)数据

C/C++深层探索/姚新颜编著.—北京：人民邮电出版社，2002.12
ISBN 7-115-10915-X

I. C… II. 姚… III. C 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字(2002)第 092498 号

内容提要

本书试图通过近 50 节的内容带领读者从各个方面去把握 C/C++的语法、语义，并通过分析 C/C++编译器生成的汇编代码，使读者明白 C/C++的某些底层实现，从而更加深入地理解 C/C++的概念、规则和不足。

本书没有面面俱到地讲述如何使用 C/C++语言编程，而是深入剖析了 C/C++语言的历史变化、各项特性及底层实现。本书试图引领读者不仅在 C/C++语言的范围内学习，而且更侧重于从汇编语言的角度、从编译程序和链接程序的角度去了解、分析 C/C++语言。通过本书，希望读者不仅能看清 C 语言的现在，还会知道 C 语言的过去，以及把握 C 语言的未来。

本书适合已经初步掌握了 C/C++的语法，希望从一个更深的层次去了解 C/C++的读者。

C/C++深层探索

◆ 编　　著　姚新颜
责任编辑　王文娟

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号
邮编　100061　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
读者热线　010-67132692
北京汉魂图文设计有限公司制作
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：800×1000　1/16
印张：19.5
字数：332 千字　　2002 年 12 月第 1 版
印数：1-5 000 册　　2002 年 12 月北京第 1 次印刷

ISBN 7-115-10915-X/TP・3234

定价：32.00 元

前　言

如果你已经初步掌握了C/C++的语法，开始渴望从一个更深的层次去了解C/C++的一些底层实现，又苦于国内很少有这方面的参考资料，那么，请翻一下这本书吧，你会有所发现的。

现在国内的计算机图书市场上，讲述C语言的书籍数量很多，但其定位主要都是面向初学者，很少能够做到深入剖析语言的历史变化、各项特性及底层实现，更不用说让读者读完之后有一种豁然开朗的感觉。

本书将引领读者不仅仅在C语言的范围内学习C语言，还要从汇编语言的角度、从编译程序和链接程序的角度去了解、分析C语言。通过本书，读者不仅仅会看到C语言的现在，还会知道C语言的过去，以及把握C语言的未来。

C/C++已经有20多年的历史，在这个过程中，编程语言的设计理念有很大的发展，作为使用最广泛的系统编程语言的C，以及作为最主要的编译型面向对象编程语言C++，都在不断变化。如何抓住C/C++的内在脉络，穿过各种表象去把握住语言的真正设计思想，是每一个认真的C/C++程序员将要面对的考验。

这本书是一次尝试。

要达到这个目标，身为作者我必须严肃对待，另外读者也要在阅读过程中不断思考、提问和自己动手寻找答案。这个世界上没有一本书会告诉你所有的事情，让你不用动脑筋光看书就真正学会知识。所以，当你阅读本书碰到困难时，千万不要轻言放弃，那些内容绝对不是你想象中那么难，花时间琢磨一下或许就会柳暗花明。

另外，本书的示例平台是GNU/Linux系统，不是我们常见的Windows系统；编译器是GCC而不是Visual C++或者C++ Builder。Linux对于相当一部分读者来说可能比较陌生，但考虑到目前我们已经有很多种途径去学习、使用Linux，我还是坚持自己的这个决定。

本书分五大部分：

Part I讲述一些C语言的基本概念；

Part II 进一步讲解 C 语言的一些难点；

Part III 为大家分析一些容易被忽略的特性；

Part IV 介绍了 C 语言的最新发展；

Part V 对 C++作了某种程度的讨论。

虽然我建议大家按部就班地从头开始看，但如果读者对某些章节的内容已经相当熟悉，则可以单独挑出感兴趣的章节进行阅读。

大多数章节都会附有充足详细的注释以帮助读者更加容易地理解正文的内容，具体的格式对应关系很简单，譬如第 03 节正文第一处需要注释的地方会用“[1]”标记，读者可以在该节后面的“[FN0301]”处找到注释。至于所有建议的参阅资料都可以在附录 A 中找到对应的条目。

最后，请容许我在这里对下面的人们表示衷心的感谢：

首先是我的父母，在整个写作过程中，他们给予我最多的帮助；其次是曾燕燕女士，她无偿地为我提供了写作过程中需要参考的所有外文资料；易峰、谌贻荣、曹文花、叶红宁、肖颖琳、湛庆延、刘睿和肖涛都在我陷入困境的时候热情地伸出友谊之手。

对本书的所有建议和批评可以发到以下电子邮箱：fred@263.net

谢谢！

编者

2002 年 8 月

目　录

Part I

00 预备知识

本书的很多部分需要通过C/C++编译器输出的汇编代码来分析C/C++的内部实现，所以读者最好能够初步熟悉i386平台的汇编指令及相关知识[1]。下面是本书中经常出现的几条汇编指令[2]：

PUSH SRC [3]

例如：

```
push  eax
push  [ebx]
push  1234
```

这条指令的实际动作是：

```
esp ← esp - 4                  //esp的值减4
[esp] ← SRC                    //把SRC复制到esp指向的内存区域
```

POP DEST

例如：

```
pop  ebp
pop  [ebx]
```

这条指令的实际动作是：

```
DEST ← [esp]                   //把esp指向的内存区域的内容复制到DEST
esp ← esp + 4                  //esp的值加4
```

MOV DEST, SRC

例如：

```
mov  eax, ebx
mov  [ebx], 1234
```

这条指令的实际动作是：

```
DEST ← SRC                    //把 SRC 复制到 DEST
```

LEAVE

执行这条指令相当于连续执行以下两条指令：

```
mov  esp, ebp
pop  ebp
```

CALL SRC

例如：

```
call  ebx
call  [ebx]
call  1234
```

这条指令的实际动作是：

```
push  eip                     //eip 的内容入栈保存
eip ← SRC                     //把 SRC 复制到 eip
                              //然后 CPU 从 eip 指向的新区域读取、执行指令
```

RET

这条指令的实际动作是：

```
pop  eip                      //把 esp 指向的内容复制到 eip，esp 的值加 4
                              //然后 CPU 从 eip 指向的新区域读取、执行指令
```

ADD DEST, SRC

例如：

```
add  eax, 1234
```

这条指令的实际动作是：

```
DEST ← DEST + SRC             //把 DEST 加上 SRC 的结果放到 DEST
```

SUB DEST, SRC

例如：

```
sub  esp, 4
```

这条指令的实际动作是：

```
DEST ← DEST - SRC          //把 DEST 减去 SRC 的结果放到 DEST
```

AND DEST, SRC

例如：

```
and  esp, -16
```

这条指令的实际动作是：

```
DEST ← DEST and SRC        //把 DEST 和 SRC 的“与”运算结果放到 DEST
```

而本书用作示例的编译器[4]所产生的汇编代码是基于 AT&T 风格，和 Win32 平台常见编译器[5]生成的 Intel 风格的汇编代码有点不同，但其实差别很小，我们只需要注意 4 点即可：

#1 汇编指令中带有后缀，指示操作数（寄存器或内存）究竟是 8 位、16 位还是 32 位[6]。

例如：

```
movb  $90, -24(%ebp)                    //操作数是 8 位的
movw  %ax, %bx                          //操作数是 16 位的
movl  %ebx, %eax                        //操作数是 32 位的
```

#2 源操作数和目标操作数的位置安排与 Intel 风格刚好相反。

Intel 风格的指令是目标操作数放在“前面”，源操作数放在“后面”：

```
mov  eax, ebx                           //把 ebx 的内容复制到 eax
```

而 AT&T 风格的指令是源操作数放在“前面”，目标操作数放在“后面”：

```
movl  %ebx, %eax                        //把 ebx 的内容复制到 eax
```

#3 寄存器名前加上“%”， 常数、符号的地址以“$”开头。

例如：

```
movl  $1234, %eax                    //把常数 1234 放到 eax
movl  $0x64, %eax                    //把常数 0x64 放到 eax
movl  $var, %ebx [7]                 //把符号 var 的地址放到 ebx
```

#4 间接寻址用圆括号表示，偏移量写在括号外面。

例如：

```
mov  eax, [ebx+4]                    // Intel 风格
movl  4(%ebx), %eax                  // AT&T 风格
```

至于我们用作实验的软件平台 GNU/Linux 系统，读者可以从网上下载，或购买光盘，而本书所用的 GCC3.2 编译器系列也可以在很多站点下载得到[8]。由于我们仅仅使用 C/C++编译器，所以只需下载其中的两个文件即可：

```
gcc-core-3.2.tar.gz
gcc-g++-3.2.tar.gz
```

把这两个文件复制到相同的目录，例如“/tmp”下，然后解压：

```
#cd  /tmp
#tar  zxvf  gcc-core-3.2.tar.gz
#tar  zxvf  gcc-g++-3.2.tar.gz
```

进入源代码的目录，然后进行配置、编译：

```
#cd  gcc-3.2
#./configure   --prefix=/opt/gcc32 [9]
#make
#make  install
#cd  /tmp
#rm  -rf  gcc*
```

如果编译成功，则在目录“/opt/gcc32/bin”下有二进制可执行文件 gcc 和 g++，这时就可以使用 3.2 版本的 GNU C/C++编译器[10]，具体安装细节请参阅相关的文档。

另外，本书一般在专业术语的第一次出现时给出它的英文原词，以提供给读者参考对照。

[FN0001]：如果你对“mov ebp, esp”之类的语句不知其意的话，最好先参考一下相关书籍再来阅读本书。

[FN0002]：这里对这些指令的描述是非常粗略的，仅仅是为了帮助读者理解其中的大概，如果需要最详细、最权威的阐述，请参阅[Intel, 2001]。

[FN0003]：SRC 是 source 的缩写，代表源操作数；DEST 是 destination 的缩写，代表目标操作数。除了个别地方，这些操作数一般都是 32 位（4 个字节）的。

[FN0004]：本书使用 GCC3.2 的 i386 版本，它使用 AT&T 风格的汇编代码。AT&T 风格在 UNIX 世界的汇编语言中被广泛使用，请参阅[Konstantin, 2001]。

[FN0005]：例如 Win32 平台上的 Visual C++。

[FN0006]：如果对操作数的确定不会产生歧义，则后缀其实也可以省略掉，例如：

```
movl  %eax, %ebx
```

可以写成：

```
mov  %eax, %ebx
```

不过，GCC 为了统一起见，所有汇编指令都加上相应的后缀。

[FN0007]：作为对比，请读者注意：

```
movl  $var, %ebx          //把 var 的地址放到 ebx
movl  var, %ebx           //把 var 的值放到 ebx
```

[FN0008]：请参阅附录 B。

[FN0009]：假设要把 GCC 3.2 安装在目录“/opt/gcc32”，以此类推。

[FN0010]：由于系统仍然按照旧的 PATH 环境变量搜索可执行文件，所以，必须先把旧的编译器文件改名，然后建立符号连接：

```
#cd   /usr/bin
#mv   gcc   gcc.old
#mv   g++   g++.old
#ln  -s   /opt/gcc32/bin/gcc   gcc
#ln  -s   /opt/gcc32/bin/g++   g++
```

01 C/C++语言的发展简史

C 语言最初是由 AT&T 贝尔实验室（Bell Labs）的 Dennis Ritchie 在 BCPL 和 B 语言的基础上设计，并在一台 DEC PDP-11 上首次实现的。简洁、高效和可移植性强使其迅速展现出作为系统级编程语言的强大生命力，随后不久，UNIX 的内核（kernel）及应用程序全部用 C 语言改写，从此，C 成为语言 UNIX 环境下使用得最广泛的主流编程语言。

1978 年，Dennis Ritchie 和 Brian Kernighan 合作推出 The C Programming Language 的第一版[1]，书末的“参考指南（Reference Manual）”一节给出当时 C 语言的完整定义，成为那个时候 C 语言事实上的标准，人们称之为“K&R C”。

随着 C 语言在多个领域的推广、应用，一些新的特性不断被各种编译器实现并添加进来，于是，建立一个新的“无歧义、与具体平台无关的 C 语言定义”成为越来越重要的事情。1983 年 ASC X3[2]成立了一个专门的技术委员会 J11[3]，负责起草关于 C 语言的标准草案。1989 年草案被 ANSI 正式通过成为美国国家标准[4]。为了区别于 K&R C，人们称之为“C89”。

这时，The C Programming Language(2nd Ed) [5]也开始出版发行，书中的内容根据当时最新的 ANSI C 进行了更新。随后，ISO 在 1990 年批准 ANSI C 成为国际标准[6]，于是“ISO C”（或称为“C90”）诞生了[7]。除了标准文档在印刷编排上的某些细节不同外，ANSI C（C89）和 ISO C（C90）在技术上是完全一样的。

然后，ISO 分别在 1994、1996 年出版了 C90 的技术勘误文档，更正一些正式标准中的印刷错误，并且还在 1995 年通过了一份 C90 的技术补充，对 C90 作了极微小的扩充，经过这次扩充后的 ISO C 称为 C95。

最近，ANSI 和 ISO 又于 1999 年通过并印刷了新版本的 C 语言标准（C99）[8]和对应的技术勘误文档。自然，这是目前关于 C 语言最新、最权威的定义。

现在，各种 C 编译器均能提供 C89（C90）的完整支持，对 C99 则提供部分支持，相信很快 C99 便会成为所有 C 编译器的标准。同时，为了兼容一些“很古老”的代码，

某些C编译器还在某种程度上支持K&R C，但编写ANSI C风格的代码才是规范的做法。

本书中的所有C代码都是用GNU C编译器[9] gcc编译的，gcc完全支持C89，并且还有一些对于特定平台实现上的扩充（称为“GNU C89”）。由于gcc在默认情况下采用GNU C89作为编译的标准，所以为了保证代码严格符合C89（C90）标准，可以使用特定的编译选项[10]，例如：

```
$gcc  -std=c89  -pedantic  example.c        或者：
$gcc  -std=c90  -pedantic  example.c
```

同时，gcc支持很多C99的特性，我们可以指示其按C99标准进行编译：

```
$gcc  -std=c99  -pedantic  example.c
```

本书的C代码究竟依据什么标准进行编译要看具体情况，一般地，如果没有特别的命令行参数，则表示用C89（C90）标准进行编译，并且暗示对于本例子而言，C89（C90）和GNU C89是完全相同的。如果例子中的编译命令带有“-std=cXX -pedantic”，则读者可以根据XX判断出编译的标准。

与C相比，C++的发展更为曲折漫长。C语言从K&R C到C89虽然经历了10年，但语言特性并没有太多的增改。C++就不一样了，从1979年Bjarne Stroustrup提出“带类的C（C With Classes）”概念[11]开始，到1998年ANSI和ISO正式通过长达700多页的C++语言标准[12]，前后整整20年。在此期间，C++的语言特性不断增加，到目前为止，C++已经成为功能最复杂、特性最丰富的系统级编程语言。

让我们简要地回顾一下这段不寻常的历史。

1979年，B.Stroustrup开始研究如何增强C语言的数据抽象能力使其最终能够实现面向对象的编程特性。他借鉴Simula语言的“Class”概念，把“类”的思想熔入到C语言中，从而产生一种具备更强的数据抽象能力的新型编程语言[13]。为什么选择C呢？原因是C语言具有高效、扩展性和移植性强的特点。B.Stroustrup希望设计出来的新语言能够成为效率非常高的主流系统编程语言，当时只有为数不多的编译型语言可供选择，比如Fortran、Pascal又或者C。Fortran是面向数值计算的，科学用途居多，显然无法担负通用型系统编程的重任；Pascal最初是基于教学用途设计

出来的，目的是通过简单的语言特性帮助使用者进入面向过程编程的大门，但也由于此，Pascal 缺乏很多支持大型系统开发（例如编写操作系统）的高级特性[14]；最后，符合要求的大概只有 C 语言了。

B.Stroustrup 当时实现新语言的做法是写一个转换程序，名字叫“Cfront”。C++代码首先通过 Cfront 翻译成普通的 C 代码，然后再用 C 编译器编译。采取这种做法是为了尽快地推广 C++，使它在各种各样的平台上能立即投入使用，因为那个时候 C 语言已经得到非常广泛的认同并在很多平台上被使用。当时如果要把 Cfront 设计为直接产生汇编代码，那就要针对每一种特定的平台设计特定的汇编生成器，这样做的成本显然太高，会阻碍 C++的传播。即使在今天，还有一部分 C++编译器同 Cfront 的工作原理是一样的[15]。

1986 年，B.Stroustrup 出版了 The C++ Programming Language 的第一版[16]。这时的 C++已经开始受到关注，使用 C++的人数达到 2000 左右。1987 年，第一次 USENIX C++会议开幕，从此，很多重要的语言特性都是在随后举行的会议上被各方面的代表（例如学术界和商业界等）提出、然后修改并由大会最后决定是否增添到 C++里面的。在这个过程中，B.Stroustrup 一直起着最重要的作用。所以，尽管 C++并不是完全由 B.Stroustrup 一个人单独设计完成的，但他仍然是人人尊敬的“C++之父（Creator of C++)”。

随着越来越多的商业 C++编译器出现在市场上，正如你猜到的那样——ANSI 开始行动了。1989 年，负责 C++标准化的 ANSI X3J16 挂牌成立。次年，B.Stroustrup 出版了他另一部经典著作——The Annotated C++ Reference Manual[17]（按照惯例，经典著作一定有简称，该著作简称 ARM)。由于当时还没有正式的 C++标准，所以 ARM 就成了事实上的标准。

值得一提的是，同样在 1990 年，C++增加了两个非常重要的新特性：Template（模板）和 Exception（异常）。前者使 C++具备了“泛型编程（Generic Programming)”的基础，后者使 C++有了更好的运行期错误处理方式。

接着，在 1991 年，ISO/IEC JTC1/SC22/WG21[18]召开第一次会议，开始进行 C++国际标准化的工作。从此，ANSI 和 ISO 的标准化工作保持同步，两者互相协调。

很多C++语言特性都是首先由J16讨论通过再提交给WG21进行审议的。

1993年，又有两个特性加入C++：RTTI（运行期类型识别）和Namespace（名字空间）。至此，C++已基本“成型”。次年，C++标准草案出台。这时，B.Stroustrup出版了一本回顾C++发展和阐述C++设计原则的著作：The Design and Evolution of C++[19]（简称“D&E”）。

本来，C++标准到此已经接近完工。但就在这个时候，STL（标准模板库）的建议草案被提交到标准委员会，立即引起充分的重视。因为如果有STL的加入，C++就成为同时具有面向对象和泛型编程的超级语言，这足以令C++的支持者们兴奋好几年。于是，对STL标准化的讨论又一次推迟了C++标准的出台。

激动人心的时刻尽管因为种种原因而迟到，但在最后——历史性的1998年，终于，ANSI和ISO先后批准C++语言成为美国国家标准和国际标准。

随后不久，在2000年，B.Stroustrup推出了The C++ Programming Language（Special Edition）[20]，书中所有内容都根据C++标准进行了更新。

从1998年底到现在，人们不断在努力。由于C++支持一大堆高级特性，从而导致实现C++的难度比较大，编写全新的C++标准库更不是一朝一夕的事情，所以直到现在（2002年）还没有任何组织敢于宣布自己的C++编译器完全支持ANSI/ISO C++标准。并且，目前很多自由操作系统平台上使用的C++编译器[21]仍然是使用老式的IO Stream库，它们对C++中Stream的实现不是基于模板的，更不用说通过实现Locale从而实现国际化编程。

不过，情况正在不断改善，最新的C++编译器（例如本书使用的g++ 3.2）都实现了全新的、基于模板的Stream库，某些领导业界技术潮流的企业甚至已经在自己支持的平台上实现了C++的国际化编程环境。相信不久之后，各种C++编译器都会完成最终的冲刺。

而在C++标准通过接近5年之际，更新、更强大的C++也已经呼之欲出，ANSI和ISO正在酝酿出炉新版本的C++标准，让我们拭目以待吧。

[FN0101]：这本书被称为“K&R”，请参阅[Kernighan & Ritchie, 1978]

[FN0102]：ANSI 属下专门负责信息技术标准化的机构，现在已改名为 INCITS

[FN0103]：J11 是该委员会的编号，所以委员会的全称是：X3J11

[FN0104]：请参阅[ANSI, 1989]

[FN0105]：请参阅[Kernighan & Ritchie, 1988]

[FN0106]：负责进行 C 语言国际标准化工作的委员会工作组是：ISO/IEC JTC1/SC22/WG14（ISO/IEC 联合技术第 1 委员会第 22 分委会第 14 工作组）

[FN0107]：请参阅[ISO, 1990]

[FN0108]：请参阅[ISO, 1999]

[FN0109]：GCC 版本号：3.2（2002 年 8 月发布）

[FN0110]：请参阅[Stallman, 2001]

[FN0111]：请参阅[Stroustrup, 1994]

[FN0112]：请参阅[ISO, 1998]

[FN0113]：1983年这种语言正式被命名为“C++”

[FN0114]：请参阅[Kernighan, 1981]

[FN0115]：请参阅[Comeau, 2001]

[FN0116]：请参阅[Stroustrup, 1986]

[FN0117]：请参阅[Stroustrup, 1990]

[FN0118]：负责C++语言国际标准化的技术委员会工作组的编号

[FN0119]：请参阅[Stroustrup, 1994]

[FN0120]：请参阅[Stroustrup, 2000]

[FN0121]：例如目前很多Linux发行版本附带的g++ 2.95.X

02 关于字节顺序

从下一节开始我们就要展开对C/C++语言的深入讨论，不过，在这之前，我们必须弄清楚一个问题：

在一个特定的硬件平台上，多字节数据是以怎样的顺序存放的？

譬如有一个32位的整数0x12345678（占4个字节），假设这个数存放在起始地址为0x04000000的内存中，那么，从0x04000000到0x04000003这4个字节内存的情况究竟是：

内存地址	数值
0x04000003	0x12
0x04000002	0x34
0x04000001	0x56
0x04000000	0x78

还是：

内存地址	数值
0x04000003	0x78
0x04000002	0x56
0x04000001	0x34
0x04000000	0x12

区别很明显：前者是数据的低字节部分存放在低地址内存，后者刚好相反——数据的高字节部分存放在低地址内存。

在现实中两种情况都有。CPU架构在这个问题上可以分为两大类，分别称为“little-endian”和“big-endian”。仅仅从名字我们就能够看出它们的差别。我们身边最常见的PC是基于IA-32[1]微处理器的，而IA-32属于little-endian类：

某些 RISC 架构的 CPU，例如 SPARC、PowerPC 等，则属于 big-endian 类。

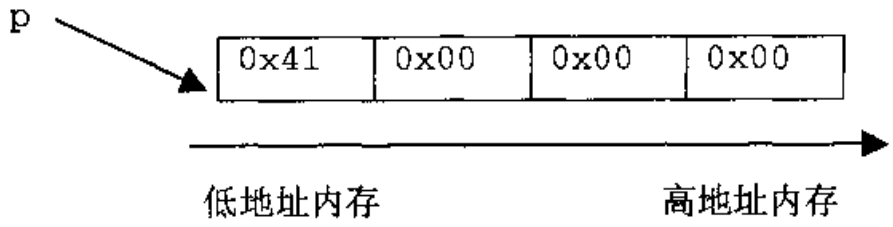
读者可能会问：这跟 C 语言有什么关系？

——别忘了，C 语言是用来编写系统程序的“中级语言”[2]，很多时候我们用它直接对硬件进行操作。稍不留神，移植性的问题就出现了。

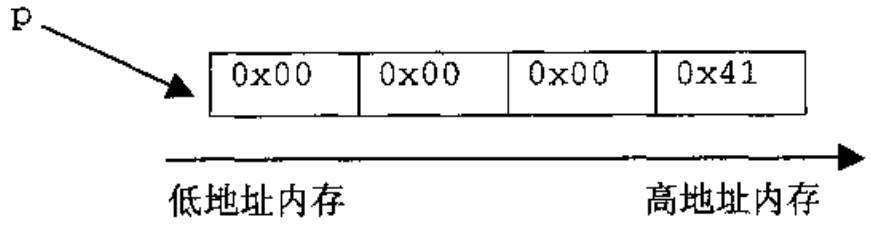
看看下面的代码[3]：

```
C01     /* Example code 02-01, file name: 0201.c */
C02
C03     #include<stdio.h>
C04     int main(void)
C05     {
C06         int i = 0x00000041;
C07         char *p = (char*)&i;
C08         printf("%s\n", p);
C09         return 0;
C10     }
```

如果在 IA-32 平台上编译执行，输出结果是大写字母“A”（字母“A”的 ASCII 码是 0x41，即 65）[4]：



而在 big-endian 平台上编译执行的话，则什么都不会输出[5]：



由于本书的所有例子都是在 IA-32[6]平台上编译执行，因此我们的讨论在某些细节方面是和 little-endian 平台具体相关的，读者在阅读时需要留意一下这个问题，不过在容易造成混淆的地方本书会作出适当的说明。

[FN0201]：从 386 到 Pentium 4 都属于 IA-32

[FN0202]：由于 C 既具有高级语言的特性，又可以像汇编语言那样进行底层操作，所以有些人把它称为中级语言。当然，这仅仅是一种说法。

[FN0203]：这里仅仅举个最简单的例子作为说明，现实中的情况更为复杂。

[FN0204]：因为数值为“0”的字节在 C/C++里面表示一个单字节字符串的结束。

[FN0205]：由于指针指向的第一个字节的值就是“0”，所以系统认为这是一个什么都没有的空字符串。

[FN0206]：本书作者的硬件平台使用 Intel Pentium III 处理器。

03 调用函数、栈

除了一些最基本的运算，我们在C语言中都是通过调用“函数（function）”来完成各种任务，这是大家都知道的。但是，“调用”函数究竟意味着什么呢？譬如下面的代码：

```
C01     /* Example code 03-01, file name: 0301.c */
C02
C03     void fun(int a, int b)
C04     {
C05         /* 这个函数什么都不做*/
C06     }
C07
C08     int main(void)
C09     {
C10         fun(100, 200);          /* 调用函数 */
C11         return 0;
C12     }
```

函数 fun 在 C10 被调用，这个调用过程究竟是怎样的？看看汇编代码[1]就知道了：

```
$gcc  -S  0301.c  [2]
$less  0301.s  [3]
```

```
A01     .globl fun
A02         .type  fun,@function
A03     fun:
A04         pushl  %ebp
A05         movl   %esp, %ebp
A06         popl   %ebp
```

```
A07         ret
A08
A09     .globl main
A10         .type  main,@function
A11     main:
A12         pushl  %ebp
A13         movl   %esp, %ebp
A14         subl   $8, %esp
A15         andl   $-16, %esp
A16         movl   $0, %eax
A17         subl   %eax, %esp
A18         movl   $100, (%esp)
A19         movl   $200, 4(%esp)
A20         call   fun
A21         movl   $0, %eax
A22         leave
A23         ret
```

A01 表示 fun 是全局可见的，A03 给出符号（symbol）fun 的位置，从 A04 开始往后就是对应原来 C 函数 fun 的代码，直到 A07 的“ret”语句从函数返回从而结束函数的执行。同样的道理，main 也是一个全局函数（global function）[4]。

不过，我们暂时不用理会这些，在这里只需要知道：

C11 调用 fun（即这面这行 C 语句）：

```
fun(100, 200);
```

对应 3 条汇编语句：

```
movl   $100, (%esp)        // 参数 a 入栈
movl   $200, 4(%esp)       // 参数 b 入栈
call   fun                 // 调用符号 fun 所在位置的代码(即 A04)
```

它们做的事情是把参数压栈[5]，然后以“call”指令调用 A04 的代码，这时指令的执行点就转到 A04。到执行完 A06 的时候，代码通过 A07 的“ret”返回，执行点又回到 A21。

说到这里，大家已经知道“调用一个函数”所发生的事情——两个步骤：

首先是参数入栈[6]（通常使用 push 或 mov 指令把参数复制到栈里面）

然后是指令的执行点转移到函数代码入口（通常用 call 指令实现跳转）

现在我们再讨论这个过程所涉及的数据结构：“栈（stack）”。

提起“栈”，第一个反应是它具有后进先出（LIFO）的特性。我们知道，一个程序由数据和代码两大部分构成，而数据有几种类别，一种是“静态”的，也就是说在整个程序运行期间，它在内存中的位置（即地址）是固定的，代码可以对其反复访问，C 语言中的外部变量（external variable）、内部静态变量（static internal variable）[7]就属于这种；另一类是“动态”的，它们在内存的位置不是固定的，例如内部（非静态）变量（internal variable）。通常，内部变量的存取就是对“栈”进行操作。以下面的代码为例，我们探讨在函数调用过程中“栈”的变化：

```
C01   /* Example code 03-02, file name: 0302.c */
C02
C03   void fun(int a, int b)
C04   {
C05       int local = a;          /* 对内部变量 local 进行操作 */
C06       a = 3;                  /* 对参数 a 进行操作 */
C07       b = 4;                  /* 对参数 b 进行操作 */
C08   }
C09
C10   int main(void)
C11   {
C12       fun(100, 200);
C13       return 0;
C14   }
```

0302.c 和 0301.c 相比其实只是在 fun 里面多了一个内部变量 local 以及几条

赋值语句，C05、C06、C07 分别对内部变量和参数进行操作，由于内部变量和参数都放在栈里面，我们可以观察一下汇编指令是如何操作内存的，借此了解栈的布局：

```
$gcc  -S  0302.c
$less  0302.s
```

```
A01         .globl fun
A02             .type  fun,@function
A03         fun:
A04             pushl  %ebp
A05             movl   %esp, %ebp
A06             subl   $4, %esp
A07             movl   8(%ebp), %eax
A08             movl   %eax, -4(%ebp)
A09             movl   $3, 8(%ebp)
A10             movl   $4, 12(%ebp)
A11             leave
A12             ret
A13
A14         .globl main
A15             .type  main,@function
A16         main:
A17             ...
A18             movl   $100, (%esp)
A19             movl   $200, 4(%esp)
A20             call   fun
A21             ...
```

进入函数代码后的一系列动作分别是：

A04，寄存器 ebp 的内容压栈保存

A05，esp 的值赋给 ebp

A06，划分出用于存放函数内部变量的整个框架[8]

到此，栈的布局如图 03-1（其中 X 代表某个内存地址）所示。

内存地址	存放的内容
...	...
X+15	b
X+14	
X+13	
X+12	
X+11	a
X+10	
X+9	
X+8	
X+7	“call fun”后面第 1 条指令的地址
X+6	
X+5	
X+4	
X+3	原先 ebp 的值
X+2	
X+1	
X	
X-1	local
X-2	
X-3	
X-4	
...	...

寄存器	值
ebp	X
esp	X-4

注意：

1. a 和 b 是在调用 fun 函数前压栈的。
2. 从 fun 返回后执行的第 1 条指令的地址由“call fun”指令自动压栈。
3. 每个 int 占 4 个字节。
4. 通常情况下，栈是往内存的低地址方向增长的，也就是说，先压栈的内容存放在高地址区域，后压入的内容放在低地址区域。

图 03-1

然后，A07 把参数 a 的值放入 eax；A08 把 eax 的值赋给 local；09 对 a 赋值；A10 对 b 赋值。这时栈的内容如图 03-2 所示：

内存地址	存放的内容	
...	...	
X+15	b	0x00
X+14		0x00
X+13		0x00
X+12		0x04
X+11	a	0x00
X+10		0x00
X+9		0x00
X+8		0x03
X+7	“call fun”后面第 1 条指令的地址	根据实际情况而定
X+6		
X+5		
X+4		
X+3	原先 ebp 的值	根据实际情况而定
X+2		
X+1		
X		
X-1	local	0x00
X-2		0x00
X-3		0x00
X-4		0x64
...	...	

注意：
这里的硬件平台是属于 little-endian。

图 03-2

最后，当 fun 做完所有事情后，A12 使执行点从函数代码返回。

[FN0301]：由于篇幅的关系本书给出的汇编代码均有所省略。

[FN0302]：这条命令是让 gcc 输出汇编代码，对应的文件名后缀是“.s”。

[FN0303]：**less** 是 Linux 上一个观看文本文件的常用命令，你可以用“↑”、“↓”、空格键等控制屏幕显示的内容，按“q”退出程序。如果你的系统上没有安装 less，则可以用 **more** 命令代替，例如：

```
more  example.s
```

[FN0304]：关于全局函数的概念在后面的章节有详细介绍。

[FN0305]：C 语言标准并没有规定参数入栈的顺序，编译器厂商可以自行决定参数是从右往左还是从左往右压栈。不过，大多数 C 编译器的参数都是从右往左入栈，即是说，靠右边参数的内存地址比靠左边参数的要高。大家分析一下汇编代码就会发现 gcc 也是这样安排的。

[FN0306]：马上我们就会观察到，gcc3.2 除了在 main 函数里调用其他函数时是通过 mov 指令传递参数并且直接用 esp 寻址外，通常都是用 push 指令传递参数，用 ebp 寻址。

[FN0307]：外部变量是指在函数外面定义的变量，例如下面的变量 a 和变量 b：

```
int a = 0;
static int b = 1;
int main(void){   }
```

内部静态变量是指虽然在函数里面定义但加上“static”修饰的变量，例如下面的变量 c：

```
int main(void){   static int c = 1;   }
```

[FN0308]：可能会有读者问：为什么一开始不直接用 esp 寻址栈里面的数据，用得着这么费功夫，先保存 ebp，然后用 ebp 存放 esp 的值，再通过 ebp 来寻址吗？

其实直接用 esp 寻址是可以的，但 gcc 默认输出的汇编代码是用 ebp 寻址，用 esp 勾画出整个函数的栈空间。这样做的好处是代码非常清晰，便于分析研究，所以本书也就按照这样的代码进行讲解。如果要追求更高的运行效率，例如在编译 Linux 内核时，

你会发现函数内部的确是直接用 esp 寻址的。可以用编译选项指示 gcc 直接用 esp 寻址，例如：　　$gcc -fomit-frame-pointer example.c

04 变量的可见范围与生存期

有了前面的基础，我们就能够很轻松地讨论变量的（可见）范围（scope）和生存期（life time）。

在一个 C 代码文件中，可以在不同的地方声明（declare）或定义（define）变量，不同地方声明或定义的变量具有不同的范围和生存期。最常见的一种情况是在函数里面定义内部变量[1]：

```
C01    /*Example code 04-01, file name: 0401.c */
C02
C03    void fun(int a)
C04    {
C05        int b = ++a;
C06        return;
C07    }
C08
C09    int main(void)
C10    {
C11        int b = 100;
C12        fun(b);
C13        return 0;
C14    }
```

C05、C11 的变量 b（注意，名字虽然相同但其实是两个不同的变量）都是在函数里面定义的内部变量。现在想一想，在函数 fun 里面能否访问到 main 里面那个变量 b，或者反过来说，在 main 里面能否存取 fun 里面定义的变量 b?

——不行。这是每一本 C 语言教科书都会告诉你的事情，但为什么不行？

我们已经知道内部变量存放在栈里面[2]，而在某个函数被调用之前编译器根本不可

能知道这个函数的内部变量的准确地址。上一节的汇编代码告诉我们，内部变量是直到执行点到达函数内部时才通过在栈里面“规划”出空间从而宣告自己的真正存在。函数一旦结束，执行点马上返回，esp 重新被调整，原先栈里面的内容已经不复存在，内部变量到了这个时候自然就不再有意义。所以，函数不可能互相访问各自的内部变量[3]。

同样的道理也适用于函数的参数。由于参数也是调用函数前才放在栈里面，函数返回后就消失，所以参数和内部变量有相同的可见范围。

另外，C12 把（main 函数的）内部变量 b 作为参数调用 fun，这是否会导致 fun（在 C05）修改 main 函数的这个内部变量呢？

不会的。调用 fun 的时候只是把 b 的值压栈，fun 并不知道 b 的地址[4]，它访问到的只是一个复制过来的副本，这个副本在 fun 里面用 a 代表。

但是，读者会进一步问：有没有变量的可见范围不仅仅局限于某个函数里面，而是所有函数都能够对其进行访问呢？

——当然有。外部变量[5]就是这样的：

```
C01   /*Example code 04-02, file name: 0402.c */
C02
C03   int global;
C04   void fun()
C05   {
C06       int local = 4;
C07       extern int global2;          /* 声明外部变量 */
C08       global = 5;
C09       global2 = 6;
C10       return;
C11   }
C12
C13   int global2;
```

```
C14    int main(void)
C15    {
C16        int global;        /* 和外部变量同名的内部变量 */
C17        global = 100;
C18        fun();
C19        return 0;
C20    }
```

C03、C13 分别定义了外部变量 global 和 global2。C08 可以直接对 global 进行操作，因为 global 的定义出现在 fun 的代码前面，所以编译器知道 global 的变量类型是 int。至于 global2 就有点不同，它也是外部变量，不过其定义点（C13）出现在 fun 的后面，所以必须有一个声明语句（注意是声明，不是定义）出现在 C07，然后 fun 才能在 C09 对 global2 进行赋值。

对 main 来说，它的前面虽然已经有 global 的定义，但由于它自己又在 C16 定义了一个内部变量（名字也叫 global），所以 C17 赋值操作的对象是作为 main 内部变量的 global 而不是 C03 所定义的外部变量 global[6]。

可以看出，外部变量不在任何函数里面定义。那编译器在哪里为它们分配空间呢？

由于外部变量具有被所有函数访问的特性，在整个程序运行期间它们的地址不能变动，因此它们不可能存放在栈里面，而是存放在数据段（data segment）[7]中。这些数据的地址在程序进行链接[8]的时候就能准确算出，并且从程序开始运行到结束都是固定的，所以任何函数都能存取这些数据。

现在来看看 04-02 的汇编代码：

```
$gcc  -S  0402.c
$less  0402.s
```

```
A01     .globl fun
A02         .type  fun,@function
A03     fun:
A04         pushl  %ebp
A05         movl   %esp, %ebp
A06         subl   $4, %esp
A07         movl   $4, -4(%ebp)
A08         movl   $5, global
A09         movl   $6, global2
A10         ...
A11
A12     .globl main
A13         .type  main,@function
A14     main:
A15         pushl  %ebp
A16         movl   %esp, %ebp
A17         subl   $8, %esp
A18         andl   $-16, %esp
A19         movl   $0, %eax
A20         subl   %eax, %esp
A21         movl   $100, -4(%ebp)
A22         call   fun
A23         ...
A24
A25       .comm   global,4,4
A26       .comm   global2,4,4
```

一眼就可看出A25、A26是第一次碰到的语句，而且A08、A09也挺有新鲜感。我们暂时可以这样理解，A25通知链接程序在数据段中分配出一块地址是4字节对齐[9]的区域，大小为4个字节，这块区域用符号“global”表示，A08对global进行操

作，其实就是对这块区域进行操作。自然，A08 的意思是把常数“5”传送到 global 所代表的内存区域，同理可以解释 A09 和 A26。

到目前为止，我们讨论了两种变量，一种是在函数里面定义的内部变量，它们动态地随着函数的调用而创建于栈中并且当函数退出时不复存在，而且，一个函数不能访问其他函数的内部变量；另一种是在所有函数体外面定义的外部变量，它们从程序开始到结束一直放在数据段中，具有固定的地址，所有函数都能存取它们。

[FN0401]：这里说的内部变量是指内部非静态变量，这些变量的声明语句中没有用 static 修饰，也称为自动（auto）变量。

[FN0402]：严格地说，应该是非静态的内部变量才存放在栈里面，后面的章节将会详细展开这个问题。

[FN0403]：虽然静态的内部变量不是存放在栈里面，但基于可视范围的限制（这属于语言的语义要求），函数之间仍然不能访问各自的静态内部变量，例如：

```
int f()                    int g()
{                          {
    static int a = 0;          static int b = 1;
    ...                        ...
}                          }
```

上面的 f()和 g()各自有一个静态的内部变量，虽然变量 a、b 不是存放在栈里，但基于变量的可视范围，f()内部的代码不能访问到变量 b，同样，g()的代码也不能存取变量 a。这种限制来自于编译器的语义分析，而正文中所讲的函数不能互访内部变量，除了基于语义的要求外还有来自于变量存储地点的限制。

[FN0404]：这里指 main 函数里面定义的那个变量 b，对于 fun 自己的变量 b，它当然知道地址。

[FN0405]：严格地说，不是所有的外部变量都能被所有的函数访问，如果变量在声明

或定义时使用 static 修饰，则不能被本文件外的其他代码访问，例如：

```
int a = 0;                    static int b = 1;
int main(void){   }           int main(void){   }
```

上面的外部变量 a 能够被所有文件的代码访问，而外部变量 b 则只能被同一文件的代码访问。后面的章节将会详细展开这个问题。

[FN0406]：在函数里面，如果内部变量的名字和某个外部变量相同，则函数认为，所有对这个名字的访问都是针对内部变量的，可以说，外部变量被同名的内部变量“屏蔽”了。

[FN0407]：这里的“段（segment）”和处理器例如 i386 的“段”在概念上是不同的。这里的段其实只是代表一块区域。因为在一个程序的整个地址空间里，既有数据又有代码，于是编译器通常分别把代码、数据集中安排在某个区域。数据段就是指专门存放数据的那个区域。

[FN0408]：“链接”的概念在后面的章节将会详细解释。

[FN0409]：“对齐”的概念在后面的章节将会详细解释。

05 变量的声明和定义

搞清楚“声明（declaration）”和“定义（definition）”之间的区别在理解C语言的过程中非常关键。

“声明”仅仅是告诉编译器某个标识符是：变量（什么类型？）还是函数（参数和返回值是什么？）。要是在后面的代码中出现该标识符，编译器就知道如何处理。记住最重要的一点：声明变量不会导致编译器为这个变量分配存储空间。

C语言专门有一个关键字（keyword）用于声明变量或函数：extern。带有extern的语句出现时，编译器将只是认为你要告诉它某个标识符是什么，除此之外什么也不会做[1]。

先来试一下：

```
C01     /* Example code 05-01, file name: 0501.c */
C02
C03     extern int a;
C04     int main(void)
C05     {
C06         extern int b;
C07         a = 1;
C08         b = 2;
C09         return 0;
C10     }
```

```
$gcc 0501.c
```

马上你会收到两条出错信息：

```
undefined reference to “a”
undefined reference to “b”
```

因为 C03、C06 仅仅是声明变量，编译器虽然知道 a 和 b 是什么类型的变量，但在链接的时候却找不到它们的地址（因为变量没有被定义，所以编译器没有为它们分配存储空间），于是就出错了。

不妨把 0501.c 编译成汇编代码看看：

```
$gcc  -S  0501.c
$less  0501.s
```

```
A01         .globl main
A02             .type  main,@function
A03         main:
A04             pushl  %ebp
A05             movl   %esp, %ebp
A06             subl   $8, %esp
A07             andl   $-16, %esp
A08             movl   $0, %eax
A09             subl   %eax, %esp
A10             movl   $1, a
A11             movl   $2, b
A12             movl   $0, %eax
A13             leave
A14             ret
```

A10、A11 表明编译器根据变量声明（C03、C06）已经知道如何处理外部变量 a 和 b，但是由于没有定义变量，所以汇编代码中没有由“.comm”开头的语句为外部变量分配存储空间，导致最后链接程序找不到有效的符号而报错。

说完“声明”，现在说“定义”。

有了对比就很容易明白，定义变量意味着不仅告诉编译器变量的类型，而且编译器同时必须为变量分配空间。定义变量的同时还可以初始化变量，例如：

```
int a = 9;
```

```
char c = 'A';
```

在函数里面定义、初始化内部变量已经详细讨论过，这里主要探讨一下外部变量的定义和初始化。

首先要搞清楚编译器在什么情况下将语句认为是定义，什么情况下认为是声明。这里给出若干原则：

#1 带有初始化的语句是定义

例如：

```
int a = 1;                  /* 定义 */
int main(void){ }
```

#2 带有 extern 的语句是声明（除非对变量进行初始化）

例如：[2]

```
extern int a;                           /* 声明 */
extern int b = 2;                       /* 定义 */
int main(void){ }
```

#3 既没有初始化又没有 extern 的语句是“暂时定义（tentative definition）” [3]

例如：

```
int a;                                  /* 暂时定义 */
int main(void){ }
```

C 语言中，外部变量只能被（正式）定义一次：

```
int a = 0;
int a = 0;                          /* 错误！重复定义 */
int main(void){ }
```

又或者：

```
int a = 0;
double a = 1.0;          /* 错误！标识符 a 已经被使用 */
int main(void){ }
```

暂时定义有点特殊，因为它是“暂时”的。我们不妨这样去理解“暂时”：

暂时定义可以出现无数次，如果在链接时系统全局空间没有相同名字的变量定义，则暂时定义“自动升级”为（正式的）定义，这时系统会为暂时定义的变量分配存储空间，此后，这些相同的暂时定义（加起来）仍然只算作是一个（正式）定义。

例如：

```
C01    /* Example code 05-02, file name: 0502.c */
C02
C03    int a;                                /* 暂时定义 */
C04    int a;                                /* 暂时定义 */
C05    int main(void)
C06    {
C07        a = 1;
C08        return 0;
C09    }
C10    int a;                                /* 暂时定义 */
```

让我们看一下汇编代码：

```
$gcc  -S  0502.c
$less  0502.s
```

```
A01      .globl main
A02          .type  main,@function
A03      main:
```

```
A04             ...
A05             movl    $1, a
A06         ...
A07
A08             .comm   a,4,4
```

A08 表示编译器只给外部变量“a”分配空间一次，尽管在 0502.c 中，C03、C04、C10 一共有 3 个暂时定义语句。

大家还记得在上一节 0402.c 中的用词吗？04-02 中的 C03、C13 应该是暂时定义（因为还没有对变量进行初始化），当时称之为定义，其实是不严谨的[4]。很快大家就可以看到真正的外部变量定义在汇编代码里是怎样表示的。

刚才说到如果没有相同名字的外部变量定义，则暂时定义会自动变成（正式）定义，那么，如果有相同名字的外部变量定义呢？很简单，这时暂时定义的作用相当于声明。

例如：

```
C01     /* Example code 05-03, file name: 0503.c */
C02
C03     int a;                          /* 暂时定义 */
C04     int a;                          /* 暂时定义 */
C05     int main(void)
C06     {
C07         a = 1;
C08         return 0;
C09     }
C10     int a = 0;                       /* 定义 */
```

C10 定义了外部变量 a，所以 C03、C04 这两个暂时定义就相当于仅仅声明 a 是 int 变量。看一下汇编代码：

```
A01     .globl main
A02         .type  main,@function
A03     main:
A04         ...
A05         movl   $1, a
A06         ...
A07
A08     .globl a
A09         .data
A10         .align 4
A11         .type  a,@object
A12         .size  a,4
A13     a:
A14         .long  0
```

A08 表示 a 是一个全局可见的符号[5]。

A09 表示在数据段分配空间[6]。

A10 指示为 a 分配的地址必须是“4 字节对齐”。

A11 告诉编译器 a 所代表的空间存放的是数据。

A12 表示分配给 a 的空间大小是 4 个字节。

A13 正式给出符号 a 的位置[7]。

A14 表示用数值“0”初始化符号 a 代表的那块（4 个字节的）存储空间。

回忆一下暂时定义对应的汇编语句：

```
.comm  a,4,4
```

然后和上面的代码作一对比，相信很容易区分暂时定义和（正式）定义在汇编代码中的表示。

[FN0501]：以下会讲到，当使用 extern 的同时又对变量进行初始化，则编译器会为

变量分配空间。

[FN0502]：虽然“extern int a = 0;”这样的语句能够通过编译，但编译器会提出警告，因为在使用 extern 的同时又对变量初始化，语意有点含糊：究竟是声明还是定义？当然，最后编译器还是把这看作定义。

[FN0503]：暂时定义是 C 语言特有的，C++没有这个概念。相反，C++推行“只定义一次”原则（one difinition rule），这导致在 C 里面是暂时定义的语句在 C++里都是（正式的）定义。所以，如果 C 程序员想自己的代码能够被 C++编译器接受，一定要注意暂时定义可能带来的问题，就是说，在代码中，最好的做法是：要么声明，要么定义，并且只能定义一次。

[FN0504]：这样做的原因是为了避免一下子堆出过多的概念造成一些读者的阅读困难，而现在大家明白暂时定义和（正式）定义之间的区别就可以了。

[FN0505]：注意本书的用词，在 C 语言里，我们把 a 称为变量；在汇编里，我们称 a 为符号（symbol）。

[FN0506]：和函数对应的汇编对比一下：函数的代码会存放在代码段。

[FN0507]：即使不对 a 代表的空间进行初始化，A13 仍然要出现，否则编译器还是不会分配空间。

06 编译和链接

平时，我们口头上并不严格区分“编译（compile）”与“链接（link）”这两个专业术语。例如我们总是说“把那个C文件编译成可执行文件……”，写成命令就是：

```
$gcc  example.c
```

这个命令马上给我们产生一个“a.out”（如果程序没有错误的话）。

实际上，整个工作至少要分四个阶段，分别由不同的程序完成：

第一阶段：由预处理程序[1]执行C源文件中的预处理指令；

第二阶段：C编译器把经过预处理的C代码文件编译成汇编代码文件；

第三阶段：汇编编译器把汇编代码文件编译成目标代码文件[2]；

第四阶段：链接程序[3]把所有目标代码链接起来产生可执行文件。

所以，编译（包括第一、二、三阶段）和链接（第四阶段）是两回事。

为什么搞得这么复杂？这是因为C语言是用来进行大型项目开发的系统语言，它必须允许很多人合作开发一个项目，所以，一个C程序可以分开由很多个源代码文件组成，不一定非得把所有代码都放在同一个文件里。这样就要求编译过程必须分成某些步骤，因为既然源代码由很多个文件组成，那就是说每个单独的源文件都不可能包含所有的程序代码，所以根本不可能要求编译器仅仅根据其中某一个C文件就直接产生可执行文件，它肯定要等到所有的代码文件都完成编译才有可能输出最后的执行文件。

事实上我们以前编写的程序就是由多个文件组成的，main 函数和其他我们自己定义的函数自然就是由我们自己来编写，其他的一些函数，例如 printf，我们一样能使用，但我们从来没有编写过printf。像printf这些函数作为C语言标准库的一部分，已经由其他人（通常是编译器厂商）写好了并编译成库文件[4]，我们的代码编译后只须和这些库进行链接就可以产生可执行文件。这个过程我们以前可能一直没有察觉，不过今后就不能再含糊了。

从前面的讨论我们大概明白C代码经过编译会产生目标代码，这个目标代码又是什么呢？还记得我们把一些 C 代码编译成汇编代码吗？仔细观察汇编文件就不难得出结

论，目标代码其实已经是以机器码的形式存在，只不过还有一些符号的地址需要链接时才能解决。举个例子：

```
C01    /* Example code 06-01, file name: 0601.c */
C02
C03    #include<stdio.h>
C04    int main(void)
C05    {
C06        extern int a;
C07        printf("a = %d\n", a);
C08        return 0;
C09    }
```

```
$gcc  -S  0601.c
$less  0601.s
```

```
A01        .section  .rodata
A02    .LC0:
A03        .string   "a = %d\n"
A04
A05        .text
A06    .globl main
A07        .type  main,@function
A08    main:
A09        ...
A10        movl   $.LC0, (%esp)
A11        movl   a, %eax
A12        movl   %eax, 4(%esp)
A13        call   printf
A14        ...
```

编译程序要做的事情之一就是把所有需要确定地址的符号（即 A10 的“$.LC0”、A11 的“a”和 A13 的“printf”）记录下来，然后链接程序在找到它们的定义点之后通过计算给予合适的地址。当所有符号都有确定的地址时，链接程序就能够产生可执行文件。如果还有符号不能确定地址（找不到定义或重复定义）链接程序就会报错。譬如这样做：

```
$gcc  0601.c
```

马上就有出错信息：undefined reference to 'a'。

因为我们根本没有定义变量 a，链接程序自然就不可能确定 a 的地址，于是出错退出，最终的可执行文件也无法生成。如果我们再写一个 C 文件如下：

```
C01     /* Example code 06-01a, file name: 0601a.c */
C02
C03     int a = 1;
```

执行命令：

```
$gcc  0601.c  0601a.c
```

外部变量 a 在 0601a.c 中得到定义，链接程序可以确定 a 的地址，顺利产生可执行文件 a.out[5]。运行一下看看：

```
$./a.out
a = 1
$
```

[FN0601]：UNIX 环境中，C/C++预处理程序的名字通常是“cpp”。

[FN0602]：UNIX 环境中，目标代码文件后缀名一般是“.o”，例如：0601.o。

[FN0603]：UNIX 环境中，链接程序的名字通常是“ld”。

[FN0604]：关于库文件的概念在后面的章节将有详细介绍。

[FN0605]：我们没有自己去定义 printf 函数，但由于 gcc 默认和标准 C 库进行链接，而标准 C 库包含有 printf 的目标代码，所以 printf 的地址也能得到确定。

07 外部变量的链接性质

和内部变量不同，外部变量被存放在数据段，这样便引出一个问题：哪些代码可以访问某个外部变量呢？在C/C++里，“链接性质（linkage）”指出了外部变量的可见范围。

外部变量的好处是可以被多个函数访问，这些函数可以分布在不同的C代码文件中，但给我们带来方便的同时麻烦也会出现。由于C语言只有一个全局空间，于是很容易发生名字冲突。试想在一个项目里，几十个程序员各自编写自己的代码，不排除有这种可能：其中某些人使用了相同名称的外部变量。后果自然就是链接出错，因为链接程序发现重复定义的外部变量。

那么，有没有办法使到一个变量既可以像外部变量那样放在数据段，又不会被其他文件的代码“看到”它的存在呢？

——只要改变外部变量的链接性质就可以。外部变量的默认链接性质是“外部的（external）”，如果把变量的链接性质改为“内部的（internal）[1]”，其他代码文件就无法“看到”这个变量。

为此，C语言有一个关键字：static，它能够把外部变量的链接性质由“外部的”改为“内部的”，就是说，其他代码文件的函数访问不到经过static修饰的外部变量：

例如：

```
C01     /* Example code 07-01, file name: 0701.c */
C02
C03     static int a = 0;
C04     int main(void)
C05     {
C06         a = 1;
C07         return 0;
C08     }
```

C03 是定义了外部变量 a，如果没有 static 修饰，那么 a 的链接性质就是“外部的”。现在我们加了一个修饰语 static，于是 a 的链接性质就变为“内部的”，只有本文件（0701.c）的代码才能访问变量 a。

很神奇吧？看看汇编代码：

```
$gcc  -S  0701.c
$less  0701.s
```

```
A01         .text
A02     .globl main
A03         .type  main,@function
A04     main:
A05         ...
A06         movl   $1, a
A07         ...
A08
A09         .data
A10         .type  a,@object
A11         .size  a,4
A12     a:
A13         .long  0
```

将上面的汇编代码跟前面定义不加 static 修饰的外部变量的例子对比，大家就可以发现，仅仅少了一条语句（“xxx”代表变量的名字）：

```
.globl xxx
```

如果有这条语句（像以前的例子），xxx 就成为全局可见的符号，于是其他文件的代码在链接时就能“看到”它；加 static 的作用就是让编译器不产生“.globl xxx”这样的语句，在这种情况下，xxx 就变成局部可见的，在链接的时候就只有本文件的代码可以存取这个变量。

上面是 static 修饰外部变量定义的情况，现在看一下 static 修饰暂时定义的情

况：

```
C01    /* Example code 07-02, file name: 0702.c */
C02
C03    static int a;
C04    int main(void)
C05    {
C06        a = 1;
C07        return 0;
C08    }
```

注意，C03 没有初始化，所以是暂时定义。

```
$gcc  -S  0702.c
$less  0702.s
```

```
A01        .text
A02    .globl main
A03        .type  main,@function
A04    main:
A05        ...
A06        movl   $1, a
A07        ...
A08
A09        .local a
A10        .comm  a,4,4
```

差别很明显，与没有加 static 修饰的情况相比，多了 A09 一行：

```
.local a
```

这一条语句通知编译器做这样的事情：使符号 a 只能被本模块的代码访问而不是以前默认情况下的全局可见。

必须弄清楚，上面的 static 仅仅改变链接性质，并不影响变量的其他特性。例如像上面的例子，加了 static 的定义依然是定义，加了 static 的暂时定义仍然是暂时定义。所以就引出接下来的讨论：万一定义与声明或者暂时定义与定义之间不一致怎么办？我指的是定义语句用了 static 而声明语句却没有用 static，或者暂时定义用了而（正式）定义没有用等。各种 C 编译器对这种问题的处理不太一样，但大部分编译器会报错，下面举出几种情况作为示例。

#1 如果在使用 static 的问题上前后不一致，则（正式）定义会和暂时定义发生冲突。

例如：

```
C01     /* Example code 07-03, file name: 0703.c */
C02
C03     static int a;                   /* 暂时定义 */
C04     int main(void)
C05     {
C06         a = 1;
C07         return 0;
C08     }
C09     int a = 0;                      /* 正式定义 */
```

C09 是定义，C03 是暂时定义，两者的链接性质不一致，相当一部分 C 编译器会报错。为保证程序的可移植性，千万别这样写。

#2 在暂时定义当中的不一致也会导致冲突。

例如：

```
C01     /* Example code 07-04, file name: 0704.c */
```

```
C02
C03     static int a;                   /* 暂时定义 */
C04     int main(void)
C05     {
C06         a = 1;
C07         return 0;
C08     }
C09     int a;                          /* 暂时定义 */
```

C03 和 C09 都是暂时定义，一个用了 static，一个没有用，存在不一致，于是编译器报错。

#3 暂时定义与声明不一致则以暂时定义为准，但编译器会发出警告[2]。

例如：

```
C01     /* Example code 07-05, file name: 0705.c */
C02
C03     extern int a;                   /* 声明 */
C04     static int a;                   /* 暂时定义*/
C05     int main(void)
C06     {
C07         a = 1;
C08         return 0;
C09     }
```

C07 修改的是 C04 所定义的 a（链接性质为内部的），因为 C04 是暂时定义，优先于 C03，所以编译器把 C07 的 a 理解为 C04 所定义的那个变量。

[FN0701]：注意，在这里，具有“内部的”链接性质的变量与内部变量是两回事，前者在函数的外面定义，存放于数据段；后者在函数内部定义，存放在栈里面。

[FN0702]：由于 C 语言标准并没有明确指出这种问题的处理方式，所以各种编译器的处理方式不完全一样，即使同样是给出警告，警告的级别也不相同。最好避免在代码中出现这种情况。

08 静态内部变量

上一节，我们用 static 改变外部变量的链接性质，现在我们继续讨论 static，不过，这一次它改变的是内部变量的存储性质（storage）。

一般情况下，内部变量都被安排放在栈里面，函数被调用时才存在，函数结束时就消失。可是，有时我们可能想定义一种变量，编译器在数据段为它分配空间，从而它的生存期是整个程序的运行时间，但这些变量同时又是属于某个函数的内部变量，所以其他函数不能访问到它。很明显，这种变量既有外部变量的存储性质（放在数据段），又具有普通内部变量的可见范围（只能被自身所在的函数访问）。能做到吗？

当然可以，只要使用 static[1]。

举个例子：

```
C01    /* Example code 08-01, file name: 0801.c */
C02
C03    #include<stdio.h>
C04    int fun()
C05    {
C06        static int a = 0;     /* 定义静态内部变量 */
C07        return (++a);
C08    }
C09
C10    int main(void)
C11    {
C12        printf("a = %d\n", fun());
C13        printf("a = %d\n", fun());
C14        return 0;
C15    }
```

编译、执行，先看看输出的结果：

```
$gcc  0801.c
$./a.out
a = 1
a = 2
$
```

如果C06没有static关键字，那么两次输出的a显然应该相同，因为每一次进入函数，内部变量a就被初始化为“0”，然后函数返回a的前缀自增值就结束了。现在有了static修饰，则内部变量a就不是放在栈而是数据段中，于是第二次调用函数fun时变量a的初值就是“1”，因此这时函数的返回值就是“2”。

也许有人会疑惑：第二次执行函数fun时，C06不是再次把a初始化为“0”吗？

实际上，静态内部变量a只会被初始化一次。看看汇编代码的片段：

```
$gcc  -S  0801.c
$less  0801.s
```

```
A01         .data
A02         .type  a.0,@object
A03         .size  a.0,4
A04     a.0:
A05         .long  0
A06
A07         .text
A08     .globl fun
A09         .type  fun,@function
A10     fun:
A11         pushl  %ebp
A12         movl   %esp, %ebp
A13         incl   a.0
A14         movl   a.0, %eax
```

```
A15         popl   %ebp
A16         ret
A17         ...
```

注意一下 A02、A03、A04、A05 和 A13、A14，会发现静态内部变量 a 在名字上被编译器做了一点“小动作”（各种编译器的具体做法不一定相同），成了“a.0”，这是因为可能会在多个函数内部定义相同名称的静态内部变量，为了区分这些变量，编译器需要给予它们不同的名字。

明显地，a.0 被放在数据段，而且由于没有“.globl”语句，因此其他文件也看不见它，否则便可能会因为在不同文件中定义了相同名称的变量而产生名字冲突。[2]

A05 告诉编译器 a.0 的初始值是“0”，这个值在编译的时候已经被存放在目标代码文件中，自然地，最后生成的可执行文件也存放了这么一个初始值。程序在加载到内存的时候，这个数据会被直接复制到内存。我们可以研究 fun 函数的代码，看看其中有没有类似下面的初始化语句：

```
movl  $0, a.0
```

——根本没有。

一进入 fun 函数，马上接着的就是 A13（变量自增），事实表明，a.0 在程序加载时已经完成了初始化。

由于静态内部变量通过这种方式初始化[3]，所以用来初始化静态内部变量的值必须是编译期便可求出的常数，像上面的“0”。

譬如说，我们可以写：

```
int f();
void g(){    int a = f();    }
```

但不能写：

```
int f();
void g(){    static int a = f();    }
```

这是因为上面用来初始化静态内部变量的值不是常数而是一个函数的返回值，而函

数的返回值不可能在编译的时候就算出来。

[FN0801]：static 在 C 语言中有很多种语义，它既可以用来改变外部变量和函数的链接性质，也可以用来改变内部变量的存储性质等，于是有些人称之为“魔术词语”。

[FN0802]：试想一下，有 2 个 C 文件：

1.c

```
int f()
{
    static int a = 0;
    ...
}
```

2.c

```
int g()
{
    static int a = 1;
    ...
}
```

由于两个变量 a 都是在函数内部定义的，所以根据可见范围的语义，它们应该互不影响，各自独立。但是编译器根据某种编码规则会赋予它们相同的名字（由于是在编译不同文件的时候赋予名字，所以编译器根本不可能知道另外一个文件中已经有相同名字的变量），例如 gcc，相应地把这两个变量的名字都改为“a.0”，要是不把“.globl a.0”去掉，则在链接的时候就会在全局空间出现两个 a.0 而造成名字冲突。

[FN0803]：不仅静态内部变量，其实所有存放在“.data”段的数据（譬如外部变量，不管是静态的或是非静态的）都是这样初始化的。

09 函数的声明和定义（上）

从 K&R C 到 C90，变化最大的可能就是函数的声明和定义。C90 一方面兼容 K&R C 的代码，另一方面又增加了很重要的特性，并提倡在实际开发中广泛地使用这些新特性。而在老规定和新准则之间的灰色地带里也潜藏了不少极其隐秘的陷阱，C 程序员一不小心就会中招。下面，我们将一步一步地阐述这个问题。

首先，函数是什么？

简单地说，函数就是一段代码，在 C 语言中，函数包含了参数、函数体和返回值这些概念。我们（通过栈）往函数中传递参数，然后函数执行自己的代码，对变量（可以是外部变量或本函数的内部变量）进行一系列的操作，做完这一切就结束返回，而返回值也通过某种方式（例如：某些寄存器）传递回来。

我们已经知道在 C 语言中调用函数：

fun(p_1, p_2, ..., p_N);

通常产生这样的汇编代码：

```
A01   push  pN
A02   ...                  /* 参数入栈 */
A03   push  p2
A04   push  p1
A05   call  fun            /* 调用代码 */
A06   add   esp, XX        /* 恢复栈指针至调用前状态，XX 代表某个值 */
```

A05 使程序的执行点跳到函数的起始地址从而使函数的代码开始执行。函数执行完毕就通过返回指令使程序执行点又回到 A6。

函数跟变量不同，编译器必须知道变量的具体类型（int、char 还是 double）才能产生正确的汇编代码，但函数却总是通过 call 指令进行调用。所以，在 K&R C 中，函数可以不声明就直接调用：

```
C01    /* Example code 09-01, file name: 0901.c */
C02
C03    int main(void)
C04    {
C05        printf("Hello,world!\n");
C06        return 0;
C07    }
```

编译器看到C05调用函数printf，但在这之前根本没有关于printf的蛛丝马迹（既没有声明更没有定义，注意：上面没有include任何头文件），然而，这样的代码却能顺利通过编译、链接。

为什么？

因为编译器看到C05，它马上可以分析出这是一个函数调用语句，因为在标识符后面带括号的语句在C语言里只有这么一种可能。于是，编译器按部就班，把参数（在这个例子中是一个字符串的起始地址）压栈，然后“call printf”。其他的事情一律不管，也没必要管。

编译器：“你不是让我以一个字符串的地址作为参数来调用 printf 吗？我给你完全照办就是了。”

程序员：“那接下来呢？”

编译器：“接下来就是链接程序的事了。”

链接程序：“只要我找到满足要求的函数代码就一切OK！有麻烦别找我。”

程序员：“……”

是的，K&R C和C90的确是这样。看看这份错误明显的代码：

```
C01    /* Example code 09-02, file name: 0902.c */
C02
```

```
C03    int main(void)
C04    {
C05        printf("100 + 200 = %d\n", sum(100));
C06        return 0;
C07    }
C08
C09    int sum(int a, int b)
C10    {
C11        return (a + b);
C12    }
```

很明显，C05 调用 sum 的时候少了一个参数，但这个程序一路顺风地通过了编译和链接：

```
$gcc  0902.c
$./a.out
100 + 200 = 1291
$
```

上面的 1291 只是某一次执行的结果，程序每运行一次结果都可能不一样。这是一个极简单的程序，相信没有人会看不出 C05 的错误，但是当程序的规模及复杂度大大增加的时候，情形就不一样了。

之所以会出现这样的事情，是因为 K&R C 不对函数的参数进行类型检查[1]，你给什么参数，它不管三七二十一，进行必要的调整后就压栈，接着就调用函数。为了更好地说明问题，我们先回顾一下历史。

在第一版的 K&R 里，基本数据类型只有 4 种：[2]

```
char
int
float
double
```

前两个属于整数类，后两个属于浮点数类。对于函数的参数，K&R C 是这样规定的：

#1 所有传给函数的参数（可以是变量或常数）如果是整数类型的（例如 char）一律先扩展成 int 再入栈。

譬如说，在某个平台上，int 是 16 位的（即占用 2 个字节），那么，无论是函数调用者还是函数本身，统统都把任何整数类参数在栈里占用的空间按 2 个字节计算。

举个例子，如果有这样的函数调用：

```
char a;
...
fun(a);
```

由于 a 是 char 变量（占 1 个字节），基于上述原则，编译器首先（在某个地方，譬如寄存器）把 a 扩展成 int 类型（占 2 个字节），然后再压栈、调用 fun 函数。

而在 fun 内部，编译器通过函数定义（这个定义的形式和我们现在普遍使用的不同，我们一会再详细讨论）：

```
fun(a)                          /* K&R C 风格的函数定义*/
char a;
{...}
```

同样知道传过来的参数 a 本来是 char 变量，只不过现在已经被扩展成 int，函数 fun 完全可以在必要的时候把 a 转换回 char 类型。

#2 所有传给函数的参数（可以是变量或常数）如果是浮点数类的，一律先扩展成 double 再入栈。

譬如 double 是 64 位的（即占用 8 个字节），那么，无论是函数调用者还是函数本身，统统都把任何浮点数类参数在栈里占用的空间按 8 个字节计算。

举个例子，如果有这样的函数调用：

```
float a;
...
fun(a);
```

由于 a 是 float 变量（占 4 个字节），基于上述原则，编译器首先（在某个地方，譬如寄存器）把 a 扩展成 double，然后压栈再调用 fun 函数。

而在 fun 内部，编译器通过函数定义：

```
fun(a)                          /* K&R C 风格的函数定义*/
float a;
{...}
```

同样知道参数 a 本来是 float 变量，只不过已经被扩展成 double，它完全可以在必要的时候把 a 转换回 float。

K&R C 为什么对函数的参数作出这些默认的扩展规则呢？很简单，因为这样会简化编译器的处理工作。既然 char 会扩展成 int，float 会扩展成 double，而指针类型通常和 int 的长度是相同的，也就是说，在 K&R C 的函数里面，编译器实际上只需要处理两种数据类型：int 和 double。

#3 凡是没有指明类型的（包括返回值和参数）就当作是 int 类型。

看看前面举的两个例子，fun 的定义都没有包括返回值的类型，编译器就认为 fun 返回 int。又例如：

```
fun(a, b, c)                    /* K&R C 风格的函数定义*/
char a; double c;
{...}
```

这个函数的名字是 fun，返回值是 int，参数有 3 个：a（类型是 char）、b（代码没有指出类型，于是编译器默认为是 int）和 c（类型是 double）。

有了这 3 条规则，在 K&R C 里面几乎不需要预先声明函数，除非你要调用的函数的返回值是浮点数类型（float 或 double）。例如：

```
C01    /* Example code 09-03, file name: 0903.c */
C02
C03    int main(void)
C04    {
C05        printf("1.0 + 2.0 = %f\n", sum(1.0, 2.0));
C06        return 0;
C07    }
```

```
C01    /* Example code 09-03a, file name: 0903a.c */
C02
C03    double sum(a, b)          /* K&R C 风格的函数定义 */
C04    double a, b;
C05    {    return (a + b);     }
```

注意上面是两个 C 文件，用以下命令编译、执行：

```
$gcc  0903.c  0903a.c
$./a.out
1.0 + 2.0 = -1.998970
$
```

问题出现了。因为如果不预先声明函数返回浮点类型值，则编译器默认函数的返回值是 int 类型，于是当它看到 0903.c 的 C05，编译器就把语句理解为：printf 的第 2 个参数是函数 fun 的返回值，而因为没有特别的声明，所以这个返回值是一个 int。

但是，系统返回整数和返回浮点数所用的寄存器根本不一样。在编译 0903a.c 的时候，编译器知道 fun 返回 double 值，它肯定会把这个返回值放到浮点寄存器里面。但在编译 0903.c 的时候，编译器认为 fun 返回的是 int，于是它把约定用来返回整数的那个寄存器的值压栈，当然是大错特错。

所以，即使在 K&R C 中，0903.c 也必须修改为：

```
C01    /* Example code 09-03p, file name: 0903p.c */
C02
C03    int main(void)
C04    {
C05        double sum();          /* 指出 sum 返回 double */
C06        printf("1.0 + 2.0 = %f\n", sum(1.0, 2.0));
C07        return 0;
C08    }
```

修改后的编译命令是：

```
$gcc  0903p.c  0903a.c
```

C05 是一个函数的声明，它指出 sum 函数的返回值类型是 double。函数声明放在 main 函数的外面（例如 C02）也可以，总之，必须在调用 sum 函数之前让编译器看到这个声明。

你可以看出，C05 的函数声明没有包含任何参数的类型信息，只有光秃秃一对括号，这就是 K&R C 的函数声明——括号里没有任何字符！于是，K&R C 对函数的参数无法进行检查。

为了改善这种情况，C90 给 C 语言增加了很重要的一项特性[2]——允许函数的声明带有参数说明。例如：

```
C01    /* Example code 09-04, file name: 0904.c */
C02
C03    int fun(char, int, double);
C04    int main(void)
C05    {
C06        fun('A', 100, 4.0);
C07        return 0;
C08    }
```

现在，C03 那对括号里面有了很重要的信息：

fun 要有 3 个参数才能调用，且参数的类型分别是 char、int 和 double。如果我们传递的参数个数不对，编译器将会报错。例如：

```
fun('A', 100);        /* 错误，这行代码不能通过编译 */
```

这种带有完整参数列表（参数名字可以省略）的函数声明叫做“函数原型声明（function prototype declaration)”。它提供了一种检查机制，使编译器能够帮助我们正确调用函数。

一定要注意的是，编译器调用有原型声明的函数时不再根据 K&R C 的规则对参数进行调整，而是仅仅依照具体平台的数据对齐要求[4]进行参数压栈，例如：

```
float f = 1.0F;
...
fun(f);
```

上面的 fun 函数没有原型声明，所以编译器会依照 K&R C 的惯例先把 f 提升为 double 类型（占 8 个字节）再压栈。

如果这样写：

```
int fun(float);                    //原型声明
float f = 1.0F;
...
fun(f);
```

则编译器由于看到原型声明，不再按照 K&R C 的老规则行事。在 32 位平台上，通常 f 会被直接压栈。

因此，我们不能把 C90 引进的原型声明和老式的 K&R C 函数定义搭配在一起使用，例如：

0905.c

```
int fun(float);
int main(void)
{
    float f = 1.0F;
    fun(f);
    return 0;
}
```

0905a.c

```
fun(a)
float a;
{
    ...
}
```

上面的代码有什么问题？

关键是 0905a.c 里面的函数定义是 K&R C 风格的，编译器看到这种风格的函数体时，自然会假定调用这个函数的时候参数会按照 K&R C 惯例进行调整。在例子中，编译器当然知道 a 是 float 型，但它却以为在栈里面的是已经提升为 double 型的参数（占 8 个字节）；而实际上，在 0905.c 中，由于有了原型声明，参数 f 的的确确是作为一个 float 被压栈的（只有 4 个字节入栈）。这样，不用说你也可以想到发生什么事情——乱套了。

同样的道理，如果没有原型声明就调用函数，而函数又是用 C90 风格定义的，混乱一样会出现，例如：

0906.c

```
int main(void)
{
    float f = 1.0F;
    fun(f);
    return 0;
}
```

0906a.c

```
int fun(float a)
{
    ...
}
```

这一次，调用 fun 的时候由于没有看到原型声明，于是编译器认为这是 K&R C 代码，f 会被先提升为 double 再压栈（8 个字节入栈）；而 0906a.c 的函数定义是 C90

风格，编译器自然认为参数 a 是没有经过提升的 float 类型（栈里的参数只占用 4 个字节）——又乱套了。

而且，即使在 0906.c 的最前面多写一句：

```
int fun();
```

也不能解决问题，因为括号里没有任何东西的函数声明属于 K&R C 风格的声明[5]，看到这种声明编译器仍然会按照“老规矩”办事。

[FN0901]：应该说，K&R C 根本就没办法检查。你说它怎么检查呢？函数的定义是允许放在其他文件里面的，在引入函数原型之前，这个问题是无法解决的。而 C90 虽然有了函数原型从而可以对函数进行类型检查，但 C90 为了兼容 K&R C 的代码，在没有原型声明的情况下照样不会对函数进行类型检查。没有原型声明下编译器惟一会对函数进行参数检查的情况是函数的定义在（同一文件中）调用函数之前就已经给出。

[FN0902]：请参阅[Kernighan & Ritchie, 1978]

[FN0903]：这个特性是从 C++“引进”的。

[FN0904]：后面的章节会介绍数据对齐的概念，这里只需知道，即使有了原型声明，函数的参数照样会调整，但这种调整仅仅是为了把参数的大小凑成某些数字，例如 4 个字节等。举个例子：

```
int fun(char);
fun('A');
```

上面尽管有原型声明，但入栈的绝对不是一个字节，在 32 位平台上，编译器一般会先把参数“A”扩展成 0x00000041（即在前面补零）再压栈。这种扩展只是为了对齐的需要，和 K&R C 的调整规则是两回事。

[FN0905]：在 C90 里面，如果要表示一个没有任何参数的函数原型，则一定要加上 void，例如：

```
int fun(void);              /* fun 没有任何参数 */
```

10 函数的声明和定义（下）

函数原型的引入虽然对C程序员帮助很大，但仍然需要我们自觉去配合。

首先，不像C++，C90并没有强迫程序员必须使用原型声明[1]，因为C90要保持对K&R C的兼容。早期的C代码都是按照K&R C编写的，自然不可能有函数原型声明，累积到C90出台的时候，这些代码的数量已经相当可观，为了让这些代码（在新代码完成之前）继续工作一段时间，C90还是保留了两道缺口。

第一道缺口：程序员可以不预先声明就调用函数。

这时，相关的规则、默认约定和上一节讲的K&R C一样。

第二道缺口：K&R C风格的函数声明仍然起作用。

由于要兼容以往的代码，括号里面不带任何东西的函数声明继续发挥“余热”。

例如下面的代码虽然有明显错误，但却毫无困难地通过了编译：

```
C01    /* Example code 10-01, file name: 1001.c */
C02
C03    int main(void)
C04    {
C05        int fun();              /* K&R C风格的声明 */
C06        fun(100, 200);          /* 错误，fun根本不应有参数 */
C07        return 0;
C08    }
C09
C10    int fun() {  return 0; }
```

C06的两个参数明显是多余的，但根据编译器对C05的理解[2]，程序被当作是旧式代码得以通过。因为要声明一个不带任何参数的函数原型，必须加上C90引入的新关键字“void”：

```
C01    /* Example code 10-02, file name: 1002.c */
C02
C03    int fun(void);                  /* 原型声明 */
C04    int main(void)
C05    {
C06        fun();
C07        return 0;
C08    }
C09
C10    int fun(){   return 0;   }
```

现在，编译器可以根据 C03 对 fun 的调用检查参数了，因为编译器认为 C03 是一个原型声明。如果 C06 写成这样：

```
fun(100, 200);
```

编译器就会报错。

原型声明不仅对函数调用有作用，其实它对函数的定义一样起规范作用。例如：

```
C01    /* Example code 10-03, file name: 1003.c */
C02
C03    int fun(void);
C04    int main(void)
C05    {
C06        return 0;
C07    }
C08
C09    int fun(int a) {  retrun 0;  }    /* 错误 */
```

上面的代码不能通过编译，因为 C03 和 C09 不一致，尽管 fun 根本就没有被调用过。

那么，如果函数的（多次）声明互相不一致会怎样？

正如变量那样，函数也可以多次声明，函数声明的目的只是告诉编译器以后遇到函数调用应该怎么办。这里给出若干原则：

#1 （同一地点能够看到的多个）原型声明不能有任何的不一致。

例如：

```
C01    /* Example code 10-04, file name: 1004.c */
C02
C03    void fun(void);
C04    int main(void)
C05    {
C06        void fun(int);                /* 错误 */
C07        return 0;
C08    }
```

main 里面根本就没有调用 fun 的语句，但当编译器分析完 C06 时已经看到关于函数 fun 的 2 个原型声明（C03 和 C06），由于它们存在不一致，所以这份代码不能通过编译。

#2 （同一地点能够看到的多个）K&R C 风格声明的返回值类型必须一致。

例如：

```
C01    /* Example code 10-05, file name: 1005.c */
C02
C03    void fun();
C04    int fun();                        /* 错误 */
C05    int main(void){    return 0;   }
```

代码不能通过编译，虽然都是 K&R C 风格的函数声明，但返回值类型不相同一样导致编译失败。

#3 K&R C 风格的函数声明和原型声明可以“和平共处”，前提是返回值类型必须相同。

例如：

```
C01    /* Example code 10-06, file name: 1006.c */
C02
C03    double fun();                    /* K&R C 风格的声明 */
C04    int main(void)
C05    {
C06        double fun(int, double);     /* 函数原型 */
C07        return 0;
C08    }
```

上面的代码能顺利编译，因为 C03 和 C06 的返回值类型相同。

#4 是否对函数参数进行检查以当时是否已经看到原型声明为准。

例如：

```
C01    /* Example code 10-07, file name: 1007.c */
C02
C03    double fun();                    /* K&R C 风格的声明 */
C04    int main(void)
C05    {
C06        fun();                       /* 可以通过编译 */
C07        return 0;
C08    }
C09    double fun(double, int);         /* 原型 */
```

以上代码可以顺利编译，因为 C06 调用 fun 时原型声明（C09）还未出现，于是 C06 是合法调用。

如果这样写：

```
C01    /* Example code 10-08, file name: 1008.c */
C02
C03    double fun(double, int);
C04    int main(void)
C05    {
C06        double fun();
C07        fun();                          /* 错误 */
C08        return 0;
C09    }
```

这段代码就会编译失败，因为 C07 调用 fun 时，C03 已经出现了原型声明。

#5 （同一地点能够看到的）声明（包括 K&R C 风格声明和原型声明）与函数定义的返回值类型必须一致。

例如：

```
C01     /* Example code 10-09, file name: 1009.c */
C02
C03     int fun(){  }                       /* 函数定义 */
C04     double fun();                       /* 错误 */
C05     int main(void){    return 0;    }
```

代码编译失败，因为当编译器分析完 C04 时，它既看到函数的定义（C03），又看到声明（C04），而它们的返回值类型不相同，所以报错。

#6 （同一地点能够看到的）函数的原型声明和定义必须完全一致。

例如：

```
C01    /* Example code 10-10, file name: 1010.c */
C02
C03    int fun(){  return 0;  }          /* 函数定义 */
```

```
C04    int main(void)
C05    {
C06        int fun(int);                    /* 原型 */
C07        return 0;
C08    }
```

C03 和 C06 的不一致将导致编译失败。

#7 函数定义本身也意味着一种声明，编译器按照函数定义的风格去确定如何调整参数，但无论如何，调用时的参数个数必须和定义时的一致。[3]

例如：

```
C01    /* Example code 10-11, file name: 1011.c */
C02
C03    int fun(void){   return 0;  }
C04
C05    int main(void)
C06    {
C07        fun(100);                    /* 错误 */
C08        return 0;
C09    }
```

上面的代码编译失败，因为函数定义（C03）和调用（C07）的参数个数不一致。

[FN1001]：后面的章节会讲到，从 C99 开始，原型声明是不可缺少的。

[FN1002]：注意，编译器对 C05 的判断是 K&R C 风格的函数声明，而不是函数原型！因为函数名字后面的括号里没有任何东西。

[FN1003]：为什么这里只是说参数个数必须一样呢？因为 C 语言存在类型转换：

```
void fun(int){  }
```

```
int main(void)
{
    fun(12.34);                    /* 这一句没有错误 */
    return 0;
 }
```

上面的代码中尽管在定义的时候 fun 的参数是 int 类型，而调用时 fun 的参数是 double 型常数 12.34，但这不会引起任何编译错误和警告，因为 C 编译器认为你要它先把 12.34 强制转换成整数（即 12）然后再用这个整数作参数调用 fun，而这是 C 语言允许的行为。

11 函数的链接性质

前面已经讲过，C 语言允许将源代码分布在不同的文件里面，各自独立编译，然后把所有（需要用到的）目标码文件链接起来就能够产生可执行文件。所以，和变量一样，函数也有链接性质，这关系到某个文件的代码能否调用其他文件定义的函数。

不过，对于函数来说，链接特性并不复杂，只有两种情况："外部的"和"内部的"。和变量不同的是，在函数里面不能定义函数，因为函数不可能放在栈里面，所以，如果没有特别指出，函数应该是"全局可见的"，即任何一个文件定义的函数都能在其他任何一个文件中调用。例如：

```
C01    /* Example code 11-01, file name: 1101.c */
C02
C03    void fun(void);
C04    int main(void)
C05    {
C06        fun();              /* 调用其他文件定义的函数 */
C07        return 0;
C08    }
```

```
C01    /* Example code 11-01a, file name: 1101a.c */
C02
C03    void fun(void){  }                  /* 定义函数 */
```

注意，上面有两个 C 文件，我们分别把它们编译成汇编代码：

```
$gcc  -S  1101.c
$gcc  -S  1101a.c
```

先看一下 1101.s：

```
$less 1101.s
```

```
A01         .text
A02     .globl main
A03         .type  main,@function
A04     main:
A05         ...
A06         call   fun
A07         ...
```

A06 调用 fun，但 fun 不在 1101.s 中定义。

再看看 1101a.s：

$less 1101a.s

```
A01         .text
A02     .globl fun
A03         .type  fun,@function
A04     fun:
A05         ...
```

上面是 fun 函数的汇编代码，大家注意 A02，这一句是关键，它告诉编译器把“fun”处理成全局可见的符号，这样当其他文件的代码就调用 fun 时，链接程序就可以找到 fun 所代表的代码进行链接。

然而，名字冲突的问题又开始困扰我们。函数的名字是全局性的，万一其他文件也定义了一个相同名字的函数，链接程序就会报错。不断地跟踪、处理这样的错误是非常浪费时间的，有没有办法使其他 C 文件的代码看不见本文件定义的某些函数呢？

有，static 又登场了。经过 static 修饰的函数只能被同一文件的函数调用，其他文件根本无法知道 static 函数的存在从而不能进行调用。

例如：

```
C01   /* Example code 11-01b, file name: 1101b.c */
C02
C03   static void g(void){  }
C04
C05   void fun(void)
C06   {
C07      g();
C08   }
```

C03 的定义有 static 修饰，所以上面的函数 g 只能被 1101b.c 里面定义的函数调用（在这个例子中是 fun 调用它）。其他文件的函数，例如 1101.c 的 main，只能够调用 fun 但不能调用函数 g。

看看汇编代码：

```
$gcc  -S  1101b.c
$less  1101b.s
```

```
A01         .text
A02     .globl fun
A03         .type  fun,@function
A04     fun:
A05         ...
A06
A07         .type  g,@function
A08     g:
A09         ...
```

和没有 static 修饰的 fun 相比，关于函数 g 的描述只是少了一句：

```
.globl g
```

由于没有这条语句，符号 g 就不是全局可见的，于是其他文件的代码就看不见 g。当然，g 是局部可见的，（同一文件的）fun 仍然可以调用它。

如果我们不加上 static 关键字去修饰函数，那么函数的链接性质就被默认为是“外部的”。即是说，以下定义是等价的：

```
int fun(void){ }                    /* 默认为外部链接 */
extern int fun(void){ }             /* 明确指出为外部链接 */
```

同样道理也适用于函数声明：

```
int fun(void);              /* 声明一个具有(默认)外部链接的函数 fun */
extern int fun(void);       /* 声明一个具有外部链接的函数 fun */
static int fun(vod);        /* 声明一个具有内部链接的函数 fun */
```

那么，万一声明和定义不一致会怎样？

我们来讨论一下这个问题。[1]

第一种情况：只有函数声明和函数调用，但没有函数定义。这时，虽然编译器会给出警告信息，不过仍然可以完成编译。例如：

```
C01    /* Example code 11-02, file name: 1102.c */
C02
C03    extern int fun(void);
C04    int main(void)
C05    {
C06        fun();
C07        return 0;
C08    }
C09    static int fun(void);
```

上面的代码分别在 C03、C09 对 fun 进行原型声明，由于存在不一致的修饰，编译器会发出警告。但代码还是能够通过编译：

```
$gcc  -c  1102.c
```

无论如何，编译器对 C06 的编译结果一般都是汇编指令：

```
call  fun
```

第二种情况：既有函数声明又有函数定义。这时，如果：

函数定义前有函数声明（可以有多个声明）并且（只要其中任何一个）声明用了 static 修饰，或者函数定义本身用了 static 修饰，则函数是 static 的，除此之外函数就是全局的。

例如：

```
C01    /* Example code 11-03, file name: 1103.c */
C02
C03    extern int fun(void);
C04    int main(void)
C05    {
C06        fun();
C07        return 0;
C08    }
C09
C10    static int fun(void);
C11    int fun(void){  return 0;  }
```

在 fun 的定义（C11）之前有两处声明（C03、C10），而 C10 用了 static，所以尽管由于声明不一致导致警告信息，但代码依然通过编译，并且 fun 被处理成是 static 函数。

又例如：

```
C01    /* Example code 11-04, file name: 1104.c */
C02
C03    extern int fun(void);
C04    int main(void)
C05    {
C06        fun();
C07        return 0;
```

```
C08     }
C09
C10     int fun(void){  return 0;  }
C11     static int fun(void);
```

上面的代码在C10定义fun，但此时带有static的函数声明（C11）还没出现，并且函数定义本身也没用static，所以fun被处理成全局可见的函数，编译器同时也会给出警告信息。

最后，如果我们想声明一个static函数，最好不要这样写：

```
int main(void)
{
    static int fun(void);
    ...
}
```

原因有二：

1. 编译器会误以为你想声明一个存储性质是static（就像静态内部变量那样）的函数，于是它会给你警告信息（放心，仅仅是警告），因为函数是没有存储性质的（变量才有）。

2. 如果你真的想声明一个static函数，那么应该在函数的外面去声明，这样才能保证在声明之后定义的那个函数的确被处理成static函数，例如：

```
static int fun(void);
int main(void){    ...   }
int fun(void){    ...    }          /* fun被处理成static函数 */
```

如果这样写：

```
int main(void){    static int fun(void);    ...    }
int fun(void){    ...    }                    /* fun仍然是全局可见的函数 */
```

因为声明同样有可见范围，上面在main函数里面声明fun，则这个声明只在main内部有效。当到了main的外面，编译器就看不到fun的前面有static修饰，于是fun的链接特性还是被处理成外部的。

[FN1101]：注意，下面所总结的原则只是gcc3.2的做法，当然，这种做法符合C语言标准，但不排除其他C编译器有自己的不同安排，那些不同的安排也可能是符合标准的。所以，尽量不要让代码出现多次声明或声明与定义之间存在不一致才是最好的办法。

12 使用头文件

头文件对于C/C++程序员实在是太常见的事物了，大家每天都在用，甚至一开始编程序就习惯性地敲入“#include”。可能正因为这样，很多人未必认真思考过头文件到底起什么作用。

要知道，C代码文件在经过真正的编译之前首先要进行“预处理”。你可以通过预处理指令指示预处理程序为你做很多事情，其中一件就是把某些文件“包含”进来。为什么要包含文件而不是直接写到源文件呢？原因是包含进来的内容具有普遍性，这样的话我们只需写一个头文件，以后要用的时候就把它include进来，这是大家都知道的。

不过，关键是哪些东西该放进头文件呢？

大概有以下这些：

#1 函数原型声明

前面说过很多次，在不少场合即使你不对函数声明也可以合法调用函数（合法是指语法上没问题），但难道你不想做些什么来保证一下你不会用错误的参数调用函数吗？请不要偷懒，坚持使用函数原型声明。对于使用系统编程语言的程序员，它是一个友善的伙伴。同时，我相信没有人这么笨把所有原型声明在每一个需要调用函数的文件中都写一遍。

#2 全局变量的声明

全局变量需要所有文件都能看见它们。最好的办法是把这些变量在头文件里声明一遍[1]。以后需要访问全局变量的时候，只需include这个头文件，代码就能够合法存取全局变量。

#3 自己定义的宏和类型

例如一些常数，我们可以在头文件用“#define”指令给予常数一个有意义的名字，

以后只要包含了头文件，就可以在代码中用具体名字代替生硬难记的常数。又或者我们要用到一些复合变量，例如结构体（struct）等，则可以在头文件用 typedef[2]定义它们。

相应地，这些东西不应该放在头文件中：

#1 全局变量和函数的定义

全局变量只能定义一次，如果你把“int global = 0;”这样的语句放在头文件中，则凡是包含了这个头文件的地方都定义一次全局变量 global，到了链接的时候链接程序就会报告说找到很多个 global，不知该用哪一个。函数的情形也是这个道理，因为函数的名字在全局空间里面也应该是惟一的，如果在多个文件中定义了相同名字的函数，同样会产生名字冲突而导致链接失败。

#2 static 变量和 static 函数

static 变量的定义或暂时定义如果放在头文件，则危险就会潜伏在暗处，随时出击。假设很不幸，static 变量的名字和某个全局变量一样，根据前面说过的原则，在全局变量声明与 static 变量的定义（包括暂时定义与正式定义）之间，后者优先。于是本来应该访问一个全局变量的代码最后存取的却是一个由头文件定义的、毫无用处的 static 变量。

同样的道理也适用于 static 函数。总之，static 的东西就像是个人隐私，请把它仅仅放在真正用上它的地方，头文件明显不是合适的选择。

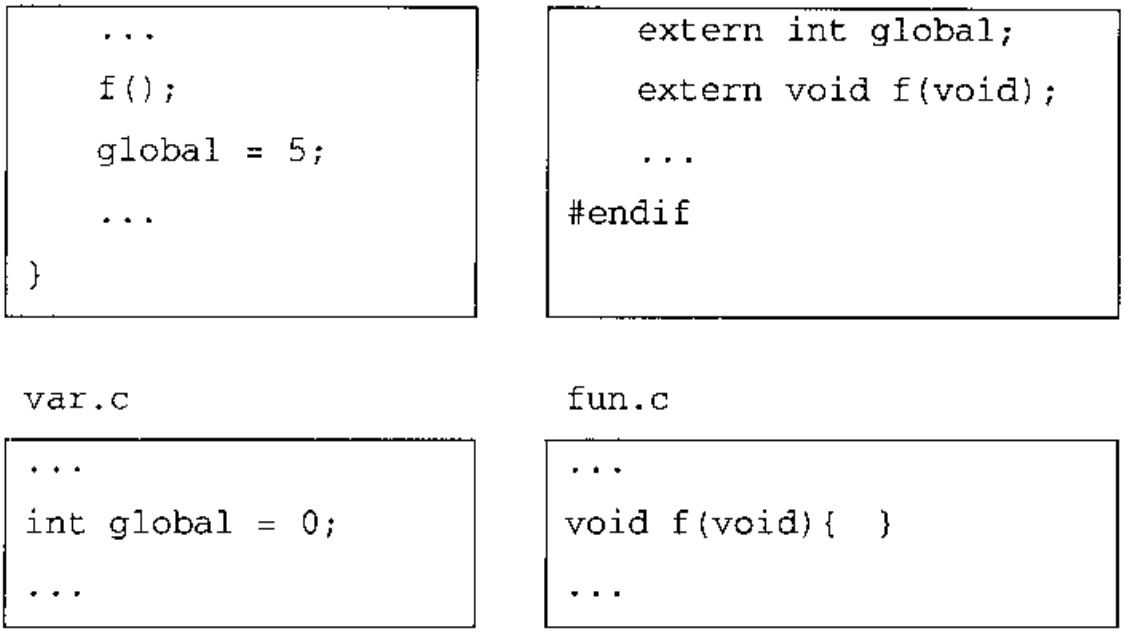
下面用一个简单的例子说明头文件的用法——全局变量和函数都应该在头文件中声明，而具体的定义则应放在 C 源文件中。请看下面的示例：

main.c

```
#include"head.h"
int main(void)
{
```

head.h

```
#ifndef HEAD_H
    #define HEAD_H
    ...
```



在这个示例中，全局变量 global 和函数 fun 的声明都放在头文件 head.h 中，而定义则分别放在 var.c（定义变量）和 fun.c（定义函数），编译命令是：

```
$gcc  main.c  fun.c  var.c
```

但你是否发现仍然有不足之处？

如何保证变量和函数的声明跟它们的定义保持绝对一致？万一在头文件这样声明：

```
extern int global;
...
```

而在 var.c 中却这样定义：

```
short global = 0;
...
```

当我们在 main.c 中访问 global 时：

```
global = 5;
```

由于头文件说得一清二楚：global 是 int，所以上面语句操作的是一段长度为 4 个字节的内存区域，可是定义 global 的 var.c 却说 global 是 short，占 2 个字节，所以结果就是紧接着 global 的 2 个字节内存区域被（意外地）修改了。如果这 2 个字节刚好存放着其他数据，那程序员就陷入了困境——“怎么这数据会莫名其妙地改变啊？！我明明没有……”

同样，函数的声明和定义如果不一致也会引起问题，所以，为了避免这种意外，应该扩大头文件的使用范围：

main.c

```
#include"head.h"
int main(void)
{
    ...
    f();
    global = 5;
    ...
}
```

head.h

```
#ifndef HEAD_H
    #define HEAD_H
    ...
    extern int global;
    extern void f(void);
    ...
#endif
```

var.c

```
#include"head.h"
...
int global = 0;
...
```

fun.c

```
#include"head.h"
...
void f(void){ }
...
```

这时，如果头文件的声明和C文件的定义不一致，编译器就会在编译的时候指出来，或者报错，或者给出警告。

[FN1201]：请注意是声明一遍，不是定义一遍。

[FN1202]：关于 typedef 的用法在后面的章节中将有详细介绍。

Part II

13 静态库

我们已经知道，C 语言允许程序员把源代码分布在很多文件中，每一个文件单独编写、编译，最后只要把需要的目标代码文件链接起来就能够生成可执行文件。

但是，如果这些（包含一个或数个函数代码的）C 文件多达几十甚至上百的时候，管理它们就变成一件令人头疼的事。你必须要记得很清楚，哪个函数在哪个文件，否则在链接的时候就会出错。上百个文件都包含了些什么函数你又怎能完全记得住？有没有一种方案，使编程者可以从一大堆目标代码文件中解脱出来？

我们可以创建一种文件，里面包含了很多函数和变量的目标代码，链接的时候只要把这个文件指示给链接程序，链接程序就自动地从文件中查找符合要求的函数和变量进行链接，整个查找过程根本不需要我们操心。

这种文件叫做“库（library）”，很形象的名字，平时我们把编译好的目标代码储存到“库”里面，要用的时候链接程序帮我们从“库”里面找出来。

其实我们一直在使用库，C 语言定义了一个标准函数集供 C 程序员使用，例如 printf 就是其中一个函数。一般地，编译器厂商把这个标准函数集的目标代码以库的形式连同 C 编译器一起提供给我们，这个库就叫做“C 语言标准函数库”，在 UNIX 世界里，它的文件名字通常是 libc。编译程序在默认情况下会把我们的代码与标准函数库进行链接[1]，于是我们就可以在自己的代码中调用这些 C 语言标准函数。

在早期，库的组织形式相对简单，里面的目标代码只能够进行静态链接，所以我们称之为“静态库（static library）”或“archive library”，它们的文件名通常是 libxx.a，xx 是长短不一的字符序列，代表库的名字，例如在 UNIX 平台，标准 C 库的文件名是 libc.a，数学函数库的文件名是 libm.a。

静态库的结构比较简单，其实就是把原来的目标代码放在一起，链接的时候链接程序根据每一份目标代码的符号表查找相应的符号（函数和变量的名字），找到的话就把该函数里面需要定位的符号进行定位，然后将整块函数代码放进可执行文件里；若是找不

到需要的函数就报错退出。

举个例子：

```
C01     /* Example code 13-01, file name: 1301.c */
C02
C03     extern int global;
C04     void f(void);
C05     int main(void)
C06     {
C07         fun();
C08         ++global;
C09         return 0;
C10     }
```

```
C01     /* Example code 13-01a, file name: 1301a.c */
C02
C03     extern void g(void);
C04     int global = 0;
C05     void f(void){  g(); }
```

```
C01     /* Example code 13-01b, file name: 1301b.c */
C02
C03     static  int local = 1;
C04
C05     void g(void){  --local;  }
```

以前，我们这样编译、链接：

```
$gcc  1301.c  1301a.c  1301b.c
```

现在，我们要做一个库文件，里面包含了 1301a.c 和 1301b.c 所定义的变量和函数，首先编译 1301a.c 和 1301b.c：

```
$gcc  -c  1301a.c  1301b.c
```

不妨用 nm 看一下 1301a.o 和 1301b.o 都有些什么[2]：

```
$nm  *.o
1301a.o:
00000010  T  f
00000004  D  global

1301b.o:
0000000c  T  g
00000004  d  local
```

容易看出，1301a.o 的符号表里面放有 2 个符号：global 和 f，其中，global 代表某个全局可见的数据（用“D”表示），链接的时候这个数据会被安排在数据段，大小为 4 字节；而 f 代表一段全局可见的代码（用“T”表示），链接的时候这段代码会被安排放到代码段，代码的长度占 16 字节（上面的“10”是十六进制）。

同理，在 1301b.o 的符号表中有 2 个符号：local 和 g，local 代表某个局部可见的数据（用“d”表示），大小为 4 字节，其他文件的代码是无法看到它的[3]；而 g 代表一段长度为 12 字节的代码。

现在我们创建一个库文件 libtest.a，把 1301a.o 和 1301b.o 的所有目标代码放进去：

```
$ar  rc  libtest.a  1301a.o  1301b.o
```
[4]

这样的话，以后我们就可以这样来编译 1301.c：

```
$gcc  1301.c  -L.  -ltest
```

上面的命令中，参数“-L.”指出库的位置是在当前目录（当前目录用“.”表示），而参数“-ltest”告诉链接程序可以在库文件“libtest.a”找到符合要求的代码和数据。

链接的过程大概如下：

首先，链接程序发现在 main 里面调用了函数 f，于是它在 libtest.a 里面查找

符号 f，结果在其中的一个模块里面找到这个符号，它的确代表一段代码，于是把这段代码复制到可执行文件并且给出一个确定的地址，再把这个地址填入调用指令[5]，然后它递归地查看 f 里面有没有需要定位的符号。在我们的例子中，由于函数 f 调用函数 g，于是链接程序继续找 g，找到后把 g 代表的代码复制到可执行文件并给出确定的地址填入调用指令。同样，在 g 代表的代码里修改了变量 local，于是链接程序又继续在库文件中寻找符号 local，然后复制、定位、填入……直到填完最后一个需要定位的符号的地址才结束链接过程，这时便可以输出可执行文件 a.out。

我们用 nm 观察一下 a.out：

```
$nm  a.out | less
...
08048464  T  f
08048474  T  g
080494e0  D  global
08048440  T  main
080494e4  d  local
...
```

可见看到，这时的 f、g、global 和 local 都有了确定的地址，在执行程序的时候，这些数据和代码就能够被准确地访问和调用。

[FN1301]：前面已经说过，当我们键入：

```
$gcc  example.c
```

后，gcc 会先后调用 cpp（预处理程序）和 ld（链接程序）完成相应的工作。而且在默认情况下，gcc 会通过命令行参数的形式指示 ld 将 example.o 与 libc 进行链接。如果你有兴趣了解详细的过程，可以加“-V”参数：

```
$gcc  -V  example.c
```

[FN1302]：nm 是 UNIX 平台上一个查看目标代码文件、可执行文件和库文件的符号表数据的命令。

[FN1303]：“无法看到它”的意思是指链接程序在为非本模块的代码查找符号时会忽略它，从而使非本模块的代码无法与它链接。

[FN1304]：有兴趣的读者可以用 nm 查看 libtest.a：

```
$nm  libtest.a
```

[FN1305]：还记得这样的汇编代码吗：

```
call  f
```

f 代表一个地址，在这个地址没有确定之前，汇编程序无法把这行汇编语句编译成最后的机器码。当我们把 C 文件编译成目标代码文件时，汇编程序仅仅是把 f 所代表的地址用适当数目的空白字节填上，然后把这些空白字节的位置记录下来。要输出可执行文件时，链接程序会（在目标代码文件或库文件中）找到 f 所代表的代码，算出适当的地址，复制，接着再回过头来把这个准确地址填入原先的空白字节。

14 动态库

上一节介绍的静态库有两个特点：

#1 链接后产生的可执行文件包含了所有需要调用的函数的代码，因此占用磁盘空间较大；

#2 如果有多个（调用相同库函数的）进程在内存中同时运行，内存中就存有多份相同的库函数代码，因此占用内存空间较多。

动态库就是为了解决这些问题而诞生的技术。顾名思义，动态链接的意思就是在程序装载入内存的时候才真正把库函数代码链接进来确定它们的地址，并且就算有多个程序运行，内存也只存在一份函数代码。[1]

由于动态库的代码必须满足这样一种条件：能够被加载到不同进程的不同地址，所以代码要经过特别的编译处理。我们把这种经过特别处理的代码叫做“位置无关代码（Position Independent Code，PIC）”。具体的实现比较复杂，这里不可能详细讲述，下面仅仅以一个很简单的例子说明其基本原理：

```
C01    /* Example code 14-01, file name: 1401.c */
C02
C03    int fun(void);
C04    int main(void)
C05    {
C06        fun();
C07        return 0;
C08    }
```

```
C01    /* Example code 14-01a, file name: 1401a.c */
C02
C03    int fun(void)
```

```
C04    {    return 0;    }
```

这里有两个 C 文件，其中，我们要把包含 fun 函数的那个文件编译成 PIC，放到动态库里去：

```
$gcc  -c  -fPIC  1401a.c
$gcc  -shared  -o  libfun.so  1401a.o
```

然后：

```
$gcc  1401.c  -L.  -lfun
```

这时已经成功生成可执行文件，不过，要使系统能够使用动态库，必须让系统知道放置动态库的位置，最简单的办法是设置 LD_LIBRARY_PATH 环境变量：

```
$export  LD_LIBRARY_PATH=/xxx/yyy/zzz
```

（/xxx/yyy/zzz 表示 libfun.so 所在的目录）

这样就可以执行命令：

```
$./a.out
$
```

尽管程序什么都没做，但确实利用了动态链接技术。我们可以用 nm 查看 a.out：

```
$nm  a.out | less
...
00000000  U  fun
...
```

在生成的可执行文件 a.out 的符号表里面清楚指出 fun 的地址还没有确定（上面的“U”表示 undefined），如果是使用静态库的话，那 fun 的地址在链接的时候就能准确算出并记录在 a.out 的符号表中，所以这充分说明了 fun 的地址的确要到程序开始运行才算出来。

当我们运行 a.out 时，载入程序知道 a.out 需要调用动态库的代码，于是它首先在内存中查找是否已经有相同的库代码在运行，如果有，则通过计算把已经在运行的代

码的地址（换算成正确的逻辑地址后）放到某张表里面，调用库代码的语句（经过一系列的过程）最终会从这张表知道库代码的地址从而能够正确调用；如果内存中没有相同的库代码在运行，则载入程序通过计算确定其逻辑地址后，再把库代码载入内存，并把逻辑地址写到那张表里，从而完成整个装载过程。

根据载入程序何时确定动态库代码的逻辑地址，可以把动态装载分为两类：

#1 静态绑定（static binding）

使用静态绑定的程序在一开始载入内存的时候，载入程序就会把程序所有调用到的动态代码的地址算出、确定下来。这种方式使程序刚运行时的初始化时间较长，不过一但完成动态装载，程序的运行速度就很快。

#2 动态绑定（dynamic binding）

使用这种方式的程序并不在一开始就完成动态链接，而是直到真正调用动态库代码时，载入程序才计算（被调用的那部分）动态代码的逻辑地址，然后等到某个时候，程序又需要调用另外某块动态代码时，载入程序才又去计算这部分代码的逻辑地址。所以，这种方式使程序初始化时间较短，但运行期间的性能比不上静态绑定的程序。

动态库已经被广泛使用，成为主流技术。我们其实已经在不知不觉地使用动态库，因为我们平时默认进行链接的标准C/C++函数库就是动态库。

[FN1401]：注意，内存中的动态库代码只有一份副本，但动态库的数据仍然可能有多份副本。因为每一个链接到动态库的进程都可能会修改库的数据，每当出现这种情况的时候，操作系统就复制出一份数据的副本，然后修改进程的地址空间映射，使它指向新的数据副本，于是进程最后修改的只是属于自己的那份数据。

15 简单类型的转换

C90 内置的简单类型只有两大类：整数类型和浮点数类型。前者包括：

char

short

int

long

后者包括[1]：

float

double

其中整数类型又可以分为两种：带符号的（signed）和无符号的（unsigned）。

一般地，在 32 位系统中这些类型的长度如下：

类型	长度（以字节计算）
char	1
short	2
int	4
long	4
float	4
double	8

其实，C 语言标准并没有规定 short、int 和 long 各占多少字节，只是要求它们满足[2]下列关系：

short 占用的空间 ≤ int 占用的空间 ≤ long 占用的空间

而按照约定，int 的大小一般刚好占用一个机器字（word）[3]，所以有上面的结果。

这样便产生一个问题：如果我们对某种类型的变量赋予其他类型的值，C 语言将怎

样处理？例如把一个 int 值赋给一个 char 变量等。

上面的问题可以进一步分成两个小问题：

#1 相同种类变量（同是整型或同是浮点型）之间的相互赋值

先讲浮点型变量。由于浮点型只有两种：float 和 double，情形比较简单。把一个 float 型值赋给 double 变量总是安全的，因为 double 是 64 位，占用空间比 float 的 32 位多出一倍，自然精度更高，可以保存的数也更大。相反则不然，把一个 double 型值赋给 float 变量就有点冒险了。精度可能会有损失，并且数值一旦超过 float 所能保存的最大值时就会发生错误。值得注意的是，编译器可能不会提示你这一点，例如：

```
float f = 1.0E+100;
```

这条语句把一个 double 型常数 1.0E+100 用作 float 变量 f 的初始值，而 1.0E+100 显然已经远远超过 float 能够存储的最大值，但是编译器依然一声不吭，它仅仅试图帮你进行转换，至于成功与否却是另一回事[4]。

不过有些时候我们还是需要类似的转换，如果我们心里百分之百清楚某个浮点数绝对可以用 float 来存储，但 C 语言默认浮点常数是 double 型的，这时可以指明浮点数的类型或进行强制转换：

```
float f = 1.0E+10F;          /* 明确指出 1.0E+10 是 float 型常数 */
```

或：

```
double d = 12.345;
float f = (float)d;          /* 把 d 的值强制转换为 float 型值赋给 f */
```

其实即使我们不明确指出，编译器同样会进行强制转换，“白纸黑字”写出来是为了提高代码的可维护性，让其他人清楚地知道：这里存在一个转换。

现在来讲讲整型变量之间的转换问题，情形稍微要复杂一点。因为整型变量即使占用同样大小的空间也存在一个“阴暗角落”——signed 还是 unsigned？

大家都知道，把 char 值赋给 int 变量，数值不会发生溢出错误，因为 int 比 char “大”。但要小心，绝大多数系统的 C 语言实现都是把 char 区分为 signed 与 unsigned，就是说，你写：

```
char c;
```

就相当于写：

```
signed char c;
```

于是编译器把 c 看成是（占一个字节的）带符号整数。看看下面的例子：

```
char c = 0x80;
int i = c;
```

你猜猜 i 的值是什么？

——“-128”！

因为 c 是 signed char，而 0x80（用二进制表示就是：10000000）的最高位是“1”，所以 c 是负数，（带符号）扩展成 32 位值就变成了 0xFFFFFF80，把这个值赋给 i，i 当然也是负数。

如果这样写：

```
unsigned char c = 0x80;
int i = c;
```

那么 i 的值就是 128，因为 c 经过（无符号）扩展后的数值是 0x00000080。

大家可以察觉出整型变量扩展的关键是对“被扩展变量”的符号解释。如果被扩展变量是带符号的（signed），那么扩展就是带符号的扩展；如果被扩展变量是无符号的（unsigned），则扩展就是无符号的扩展。

再来看看：

```
int i = 5;
int u = i;
```

有问题吗？

——绝对没有！

好，这样呢：

```
int i = 5;
unsigned int u = i;
```

——别犹豫，当然可以。编译器仅仅是把 i 的值原封不动地赋给 u，除此之外没有

任何多余的动作。

如果这样呢：

```
int i = -5;
unsigned int u = i;
.....
```

让我们仔细看一下过程：

首先，编译器把-5 转换成 32 位数值 0xFFFFFFFB 赋给 i，然后把 i 的值复制到 u，所以最后 u 仍然等于 0xFFFFFFFB。值还是那个值，但怎么解释就是另一回事。0xFFFFFFFB 对于 i(带符号整数)来说是-5，对于 u(无符号整数)来说是 4294967291。

再极端一点的例子：

```
unsigned int u = -1;
```

这好像有点前后矛盾的味道。但编译器依然“我行我素”，-1 就是 0xFFFFFFFF，于是 u 的值还是如假包换的 0xFFFFFFFF。只不过由于你把 u 作为无符号整数来使用，所以，（根据上下文）u 就被解释为 4294967295（即 2^{32} -1）。

接下来，我们再讨论一下“裁剪”。

```
int i = 4;
char c = i;
```

这样的代码意味什么？

答案更简单：编译器把 4（即 0x00000004）赋给 i，它知道 char 占一个字节，那么，把 i 的低 8 位赋给 c，于是，c 等于 4。

如果这样呢：

```
unsigned int u = 1000000;
short s = u;
```

同样道理，1000000 是 0x000F4240，short 占两个字节，于是，s 的值就是 u 的低 16 位，即 16960（0x4240）。

再来看：

```
unsigned int u = 0xFFFF00000000;
```

猜猜 u 等于多少？

——没错，等于“0”。同时，编译器会发出警告，因为 0xFFFF00000000 已经超过 unsigned int 的最大值。

#2 不同种类变量（整形与浮点型）之间的相互赋值

无论是 float 还是 double，它们所能存储的最大值都大于 4294967295（即 $2^{32}-1$），最小值都小于-2147483648（即-2^{31}），你可能会认为把一个整形值转换为浮点值一定是安全的。不过，事实并非这么简单，因为浮点型存储的数值范围虽然大但却受精度（有效数字）的限制。例如：

```
unsigned int u = 4294967295;
float f = u;
```

你猜猜结果是什么？

f 的值变成了 4294967296.000000！

因为如果要把一个数值转换成 float 再转换回来并且转换回来的数值要和原先的一模一样则要求这个数值只能有 6 位有效数字，超过 6 位有效数字就会发生一些精度的损失。所以，若非实在必要，最好不要把整数转换成浮点数处理。

反过来，如果把浮点数转换成整数呢？

我们可以预料到会发生什么事，那就是小数部分被舍掉了。只是有一点必须要注意，转换的第一步一定要先转换成以下其中一种类型：

int	long	long long
unsigned	unsigned long	unsigned long long

例如：

```
float f = 23456.789;
int i = f;
```

```
short s = i;                    /* OK */
```

这样，i 是 23456，s 也是 23456。千万不要以为 short 可以放下 23456 就一步登天：

```
float f = 23456.789;
short s = f;                    /* Error */
```

上面代码得出的 s 值根本不是 23456。[5]

最后别忘了：浮点数能够转换成整数的前提是（去掉小数部分的）数值能够用对应的整数类型表达，如果数值太大（或太小），那转换的结果也就没有意义，例如：

```
double d = 123456789876.543;
int i = d;                      /* 都溢出了！ */
```

你说，i 应该等于多少？

[FN1501]：浮点数还有一个 long double 类型，不过相当多的系统不支持超过 64 位的浮点数，在这些系统上，double 和 long double 是相同的（都是 64 位）。IA 的浮点单元支持 80 位的浮点运算，所以我们的实验平台的确存在比 double 更高精度的 long double 类型，但由于很少用到，这里不列入讨论范围（其实道理还是一样的）。

[FN1502]：除了满足这个条件，short、int 和 long 的数值范围还有相应的规定，后面的章节讲会详细讲述数值问题。

[FN1503]：IA-32 的机器字长自然就是 32 位。

[FN1504]：是否作出警告要看具体的编译器和编译选项，不过对于整数的溢出，编译器通常都会作出警告。

[FN1505]：C 语言的数值运算（包括不同数值类型间的转换）遵从国际标准 LIA（请参阅[ISO, 1994]）。而 LIA 只规定了浮点数转换为正文提到的那几种整数类型的规则。因此，类似把 float 转换成 short 这样的行为的结果是不确定的。

16 复合类型

C语言提供了几种复合类型：struct（结构体）、union（共用体）和bit field（位域）。这里不再重复它们的使用方法，只是讨论一些有趣的问题。

以struct为例，譬如：

```
C01    /* Example code 16-01, file name: 1601.c */
C02
C03    #include<stdio.h>
C04    struct s
C05    {
C06        char c;
C07        int i;
C08    };
C09
C10    struct s sVar = {'A', 5};
C11    int main(void)
C12    {
C13        printf("%d\n", sizeof(struct s));
C14        return 0;
C15    }
```

请你猜猜：程序的输出是多少？

“很简单，int占4个字节，char占1个字节，加起来就是5个字节！”

……

```
$gcc 1601.c
$./a.out
8
```

```
$
```

看到了吗，结果不是 5，而是 8。

在作出解释之前我们必须先搞清楚“alignment（对齐）”的概念。

CPU 对内存的访问并不是完全任意的，如果 CPU 想一次读（写）4 个字节，那么给出的内存起始地址最好是 4 的倍数，例如：

```
0x12345670
0x12345674
0x12345678
0x1234567C
```

大家可以看出规律就是（十六进制）地址的最低位一定是 0、4、8 或 C 的其中之一，这样的话 CPU 只需通过一次内存访问就能完成数据的存取。否则，CPU 就要通过多次访问操作来完成数据的存取。例如我们要求 CPU 从 0x12345672 读 4 个字节到寄存器 eax，CPU（很无奈地）这样做：

第一步，从 0x12345670 读 4 个字节，舍弃掉低 16 位的数据，把剩下的高 16 位数据存放到 eax 的低 16 位；

第二步，从 0x12345674 读 4 个字节，舍弃掉高 16 位的数据，把剩下的低 16 位数据存放到 eax 的高 16 位。

可以看到，如果数据不作出特别的安排，则 CPU 对数据的访问就会缺乏效率。为了提高效率，编译器一般都会把数据安放到适合它们的地址，譬如 int 占 4 个字节，则凡是 int 变量的地址都是 4 的倍数，称为“4 字节对齐”。看看这样的汇编代码：

```
.globl a
.data
.align 4
.type  a,@object
.size  a,4
a:
.long  0
```

除了那句“.align 4”，其他的我们已经明白表示什么。现在知道对齐的概念后，“.align 4”的意思就是指示编译器：分配给 a 的地址必须是 4 字节对齐的。

好，让我们返回到原先的问题。为什么 struct s 要占 8 个字节呢？

s 有一个成员 i，而 i 是 int 变量，于是 i 必须是 4 字节对齐。怎样安排 s 的大小以及地址才能保证 i 的内存地址一定是 4 字节对齐？一种最省事的办法就是把 s 安排成 4 字节对齐，然后调整（如果需要的话）i 之前的成员所占用的空间，使空间大小是 4 的倍数，这样 i 的地址到最后必然也是 4 字节对齐的。

就像我们的例子，i 的前面是 char 变量 c，由于 char 只占一个字节，所以在 c 的后面增加 3 个填充字节，凑够 4 个，从而保证了 i 的地址是 4 字节对齐。如果还有疑问的话看看汇编就更清楚了：

```
A01     .globl sVar
A02         .data
A03         .align 4
A04         .type  sVar,@object
A05         .size  sVar,8
A06     sVar:
A07         .byte  65
A08         .zero  3
A09         .long  5
```

A03：全局外部变量 sVar 的地址是 4 字节对齐的；

A05：sVar 的大小为 8 个字节；

A07：sVar.c 的初始值是 65（字符“A”的 ASCII 值）；

A08：在 sVar.c 后面填补 3 个字节，它们的值为“0”；

A09：sVar.i 的初始值是 5。

于是，我们又得到一条经验：应该小心处理 struct 的成员布局，否则编译器的某

些安排可能导致空间的浪费。例如，不要这样写：

```
struct s
{
        char x;
        int  y;
        char z;
    int  u;
};              /* s必须占用16个字节 */
```

而应该这样写：

```
struct s
{
        int  y;
        int  u;
    char x;
        char z;
};              /* s只须占用12个字节 */
```

有的读者可能会问：下面这样写的struct又会占多少字节呢？

```
struct s
{
    int i;
    char c;
};
```

“这次应该是5了！只要s是4字节对齐，那i肯定也是，至于c，反正占一个字节，根本没有对齐的需要”……

——不，事情还没完，上面的s还是占8个字节。

答案很简单，还是数据要对齐的缘故。试想一下，如果我们写：

```
struct s array[10];
```

则数组 array 由 10 个 s 组成，这 10 个 s 必须依次安排在一块连续的区域。好，如果 s 占用 5 个字节，问题就来了：array[0]当然没问题，但 array[1]就惨了——它的地址不是 4 字节对齐的！后面那些元素的地址也都全乱套了。

又例如，我们把 s 作为一个函数的参数：

```
void fun(int n, struct s sVar){   ...   }
```

参数入栈的时候，sVar 首先进栈，而编译器为了给随后压栈的 n 进行对齐，要么经过计算填充字节，要么索性把 struct s 的大小定为 8 字节。可以想象，后者更简单，实现起来更省事。

17 关于指针（上）

在 C 语言中，“指针（poiter）”的概念不太容易掌握，但其实指针并不神秘，关键是要把一些复杂的语法表达弄明白，把编译器内部实现原理搞清楚。

首先说一下语法问题。最简单的例子：

```
int *p;
```

p 是一个指针，它“指向”一个 int 值，即 p 这个变量储存的是内存中一个 int 值的地址。

从语法上怎么分析呢？

如果我说：

```
int a;
```

你马上知道，a 是一个 int 变量。

好，看看上面指针的式子，相当于[1]：

```
int (*p);
```

于是，我们认为（*p）是一个 int 变量，而“*p”的意思是把变量 p 的值当地址使用，所以我们得出结论：变量 p 的值作为地址使用，依照这个地址可以存取一个 int 值，即 p 是一个 int 指针。

如果我写：

```
int array[5];                /* 不妨写成 int (array[5]); */
```

你马上说：array 是一个有 5 个元素的数组，每一个元素都是 int。可以认为“array[5]”的意思是：array 是一个有 5 个元素的数组，每一个元素……

好，我再写：

```
int *array[5];               /* 写成 int (*(array[5])); */
```

[2]

同样，毫无困难地，array 是一个有 5 个元素的数组，每一个元素都作为地址使用，依照这个地址可以存取一个 int，就是说，array 是一个由 5 个 int 指针组成的数组。

如果有：

```
int fun();              /* 写成 int (fun());并且与 int a;作对比 */
```

不妨这样看，fun()“是”一个 int[3]，由于标识符 fun 后面有括号对“()”，所以 fun 是一个函数，就是说，fun 是一个函数，它的返回值类型是 int。

如果这样写：

```
int *fun();                 /* 写成 int (*(fun())); */
```

还是那么回事，fun 是一个函数，它的返回值被当作地址来使用，根据这个地址可以存取一个 int 值，即 fun 是一个返回 int 指针的函数。

要是这样写：

```
int (*fun)();               /* 与 int fun(); 作对比 */
```

我们可以说，(*fun) 是一个返回 int 值的函数，也就是说，fun 储存的是一个地址，依照这个地址可以得到一个返回 int 值的函数，即是说，fun 是一个函数指针，它指向一个返回 int 的函数。

继续：

```
int (*array[5])();          /* 写成 int (*(array[5]))(); */
```

一点都不难。由于(array[5])是一个函数指针，指向一个返回 int 的函数，所以，上面式子的意思是 array 是一个有 5 个元素的数组，每一个元素都是一个函数指针，指向一个返回 int 的函数。

看看下面这个：

```
int (*fun(int))(int);       /* 写成 int (*(fun(int)))(int); */
```

虽然看上去复杂，不过仍然可以按照前面的方法分析。与

```
int (*f)(int);
```

对照，一眼就看出(fun(int))是一个函数指针，指向带有一个 int 参数、返回值是 int 的函数。而 fun 本身是带有一个 int 参数的函数，于是得出结果：fun 是带有一个 int 参数的函数，它的返回值是一个函数指针，这个指针指向带有一个 int 参数并返回 int 的函数。

大家知道，声明一个函数指针然后赋给它适当的值（函数入口地址）就可以用这个指针调用函数，例如：

```
C01    /* Example code 17-01, file name: 1701.c */
C02
C03    void fun(void){  }
C04
C05    int main(void)
C06    {
C07        void (*pf)(void);
C08        pf = fun;              /* 把 fun 的地址赋给 f */
C09        (*pf)();               /* 调用 fun() */
C10        return 0;
C11    }
```

C08 中，fun 作为函数的名字，代表着函数的地址，可以赋给函数指针 pf。C09 的意思是通过函数指针 pf 调用函数，其实这样写也可以：

```
pf();
```

编译器对两种写法均认为是调用 pf 指向的函数。

上面的例子是已经定义某个函数，然后通过函数指针调用，如果我们需要直接调用某个用常数来表示地址的函数呢？譬如，我们想调用从地址 0x12345678 开始的某段代码（这段代码的返回值假设是 void），然而在 C 语言里，我们不能直接这样写：

```
(*0x12345678)();
```

因为对于 0x12345678，编译器认为它是一个 int 常数并不是一个函数地址，这时就要借助类型强制转换：

```
(*(void(*)())0x12345678)();
```

这个表达式没有你想象中那么复杂，它先把 0x12345678 强制转换成函数指针类型值，然后调用指针指向的函数。让我们一步一步地分析：

如果写

```
int a;
```

你立即说 a 是一个 int 型的变量，可见，在变量声明中把变量名字去掉剩下的就是变量的类型。我们要知道 a 是什么类型的变量，只需在 a 的声明中把 a 去掉，在这个简单的例子中去掉了 a，剩下的是 int，所以 a 是 int 变量。

接着再写：

```
(int)b;
```

你又轻松地说式子的意思是把变量 b 强制转换成 int 类型。

现在我写：

```
void (*f)();
```

大家都知道 f 是一个函数指针变量，它的类型是什么？

——（把 f 去掉）就是 void(*)()。

然后把常数 0x12345678 强制转换成 void(*)()类型，只要写：

```
(void(*)())0x12345678
```

这样我们就得到一个函数指针值，可以用它来调用函数：

```
(*(void(*)())0x12345678)();
```

[FN1701]：注意“*”是右结合的运算符，即：int *p;（又或者 int* p;）在语法上都相当于 int (*p);

[FN1702]：注意“[]”的优先级比“*”高，所以 array 先和“[]”结合，得出的结果再和“*”结合，这个顺序一旦搞错语义便完全不同。

[FN1703]：不用惊讶，“fun()”的类型（看清楚，是“fun()”的类型，不是 fun 的类型，fun 的类型是函数指针）就是 int，譬如你写：int a = fun(); 完全正确。

18 关于指针（中）

这一节我们讨论指针与数组的关系，初学者经常混淆它们的概念。

的确，指针和数组在用法上比较相似，例如：

```
int array[5];
array[3] = 1;                    /* 一般都这样写 */
*(array + 3) = 1;                /* 这样写也行 */
```

或：

```
int *p = XXX;                    /* XXX 代表某个具体数值 */
*(p + 3) = 1;                    /*  通常的写法 */
p[3] = 1;                        /* 完全可以这样写 */
```

之所以两种写法都可以，一是由于运算符“[]”和“*”是等价的，就是说你写：

```
X[i]
```

和写：

```
*(X + i)
```

是完全一样的，编译器赋予它们相同的解释；

二是由于指针和数组的名字都代表某种变量的地址。不同的是，指针是变量，所以编译器会为指针分配存储空间，并且指针的值可以在运行期随意改变，例如：

```
int a = 0;
int b = 1;
int *p = &a;              /* 编译器为 int 指针 p 分配空间并初始化 */
(*p) = 4;                 /* 对 p 指向的数据进行操作 */
p = &b;                   /* 改变 p 的值，使它指向另外一个 int 值 */
*p += 3;                  /* 对新指向的数据进行操作 */
```

而数组名字只是代表某段存储空间的起始地址，编译器不会为数组名字本身分配空

间，所以数组名字就相当于一个常量，我们不能改变数组名字代表的地址值，例如：

```
int array[5];          /* 分配 5 个 int 的空间，array 代表起始地址 */
int main(void)
{
    array[3] = 1;              /* 可以通过 array 访问数组元素 */
    array = XXX;               /* 错误，因为 array 不能被改变 */
}
```

这个道理就相当于我们不能写：

```
int a = 1;
&a = XXX;                      /* Error!  */
```

我们可以改变 a，但不能改变 a 的地址，因为给变量分配地址是编译器的事情，我们没有任何理由去干预。比如说分配给 a 的地址是 0x12345678，那么上面错误的式子相当于：

```
0x12345678 = XXX;
```

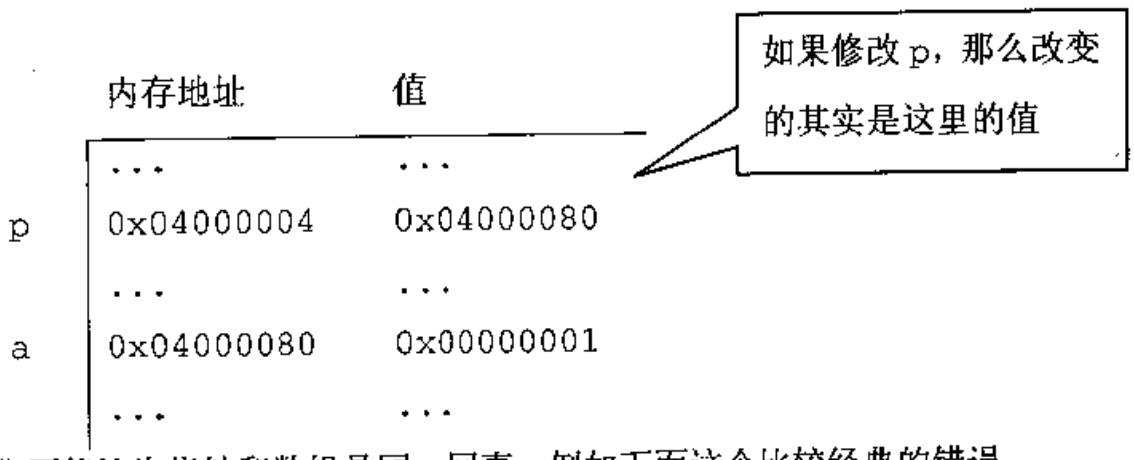
这显然没有意义。

同样是代表着某个地址，为什么我们可以改变指针呢？因为指针是变量，它在内存中有自己的空间，我们改变的其实是存储在这空间里的值。

例如：

```
int a = 1;
int *p = &a;
```

如下所示：



所以，我们不能认为指针和数组是同一回事，例如下面这个比较经典的错误：

```
C01   /* Example code 18-01, file name: 1801.c */
C02
C03   extern int *array;              /* 声明指针 */
C04   int main(void)
C05   {
C06      array[0] = 1;               /* 用指针访问数据 */
C07      return 0;
C08   }
```

```
C01   /* Example code 18-01a, file name: 1801a.c */
C02
C03   int array[5] = {0};            /* 定义的却是数组 */
```

上面代码虽然明显有问题，但（可怕的是）gcc 允许它通过编译[1]：

```
$gcc  1801.c  1801a.c
$
```

但如果执行程序就会失败：

```
$./a.out
Segmentation Fault
$
```

为什么 gcc 捕捉不到错误呢？因为上面我们把指针（声明）和数组（定义）分布在两个源文件中，于是编译器无法在编译期间检测出错误。如果我们再看一下编译器输出的汇编代码就清楚了：

```
$gcc  -S  1801a.c
$less  1801a.s
```

```
A01      .globl array
A02          .align 4
```

```
A03         .type  array,@object
A05         .size  array,20
A06     array:
A07         .long  0
A08         .zero  16
```

在汇编文件中，array 和普通的全局变量完全没有两样（A01），留意它的 5 个元素都已被初始化为“0”（A07，A08）。可以想象，进行链接后， 1801.c 中的 int 指针 array 的值就是 1801a.c 中的数组元素 array[0]的值，也就是“0”——空指针！用空指针进行访问自然导致程序运行期错误“Segmentation Fault”。

那么，难道指针和数组名字就永远不能划等号？

也不是，在某些情况下，例如函数的参数里面，它们就是完全一样的：

```
int fun(int array[]);
```

和

```
int fun(int *array);
```

是相同的声明，并且我们可以“修改”数组名字的值：

```
int fun(int array[])
{
    ...
    array = XXXX;                          /* OK */
    ...
}
```

因为在参数压栈后，数组名字相当于有了自己的存储空间（在栈里面），于是我们就能像修改指针那样去改变（栈里面的）数组名字。打个比方：

我们不能写：

```
5 = 10;                          /* 不能把 10 赋给常数 5 */
```

但如果这样写：

```
int fun(int a)
{
    ...
    a = 10;                 /* 没问题，修改参数而已 */
    ...
}
fun(5);                     /* 把常数 5 压入栈，完全可以！ */
```

则谁也不会认为有问题。

事实上，C/C++编译器的确会把参数里面的数组由数组类型转换成指针类型，也就是说，无论我们这样写：

```
void fun(int a[]){   ...   }
```

还是这样写：

```
void fun(int a[10]{   ...   }
```

C/C++编译器在 fun 里面都会把 a 转换为指针类型[2]！举个例子大家就会明白：

```
C01     /* Example code 18-02, file name: 1802.c */
C02
C03     #include<stdio.h>
C04     void fun(int a[10])
C05     {
C06         printf("%d\n", sizeof(a));
C07     }
C08     int main(void)
C09     {
C10         int array[10];
C11         printf("%d\n", sizeof(array));
C12         fun(array);
```

```
C13         return 0;
C14     }
```

```
$gcc  1802.c
$./a.out
40
4
$
```

C10 的 sizeof(array)打印出来的值是 40，这毫无疑问，因为 array 是一个有 10 个 int 的数组，所以大小就是 40 个字节。但 C11 把 array 作为 fun 的参数时，在 fun 内部（C05）打印出来的 sizeof(a)却是 4 字节，刚好是一个指针占用内存的大小。这充分说明了：在 fun 内部，参数 a 的类型是指针而不是数组。

[FN1801]：下面会讲到，这不是 gcc 的错。

[FN1802]：多维数组同样如此，例如：

```
    void fun(int a[][10]){   ...   }
```

会被 C/C++编译器转换为：

```
    void fun(int (*a)[10]){   ...   }        /* a 是一个指针 */
```

下一节会详细论述多维数组与数组指针的关系。

19 关于指针（下）

一维数组与指针的关系相信大家非常熟悉，但是对于多维数组，情况就稍微复杂一点。首先，多维数组的元素本身也是数组，这样的话，在声明或定义的时候，我们必须让编译器知道元素的大小，例如：

```
int array[2][3];
```

上面的定义指出 array 是一个二维数组，它有 2 个元素，每个元素都是有 3 个 int 的数组。也就是说，array[0]和 array[1]的类型都是 int [3]。

想想看，能不能这样写：

```
int array[2][];
```

——显然不行。因为编译器只知道元素是 int 数组，却不知道作为元素的 int 数组的大小，从而无法对 array 进行任何操作。

编译器如何寻址多维数组的元素呢？下面以二维数组为例说明一下。假设我们定义：

```
int array[3][5];
```

那数组的存储分布如图 19-1 所示（x 代表内存某个地址）：

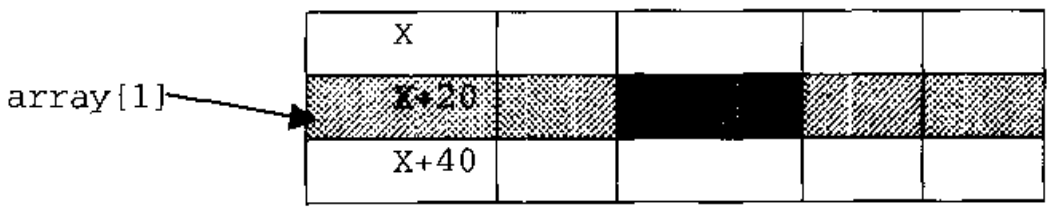


图 19-1

array[1][2]的地址是怎么算出的？

编译器首先算出 array[1]的地址，因为 array 的元素是有 5 个 int 的数组（大小为 20 字节），所以，array[1]的地址是（X+20）。然后，由于 array[1]是有 5 个 int 的数组，所以 array[1][2]的地址就是（(X+20) + 2*4），即（X+28）。

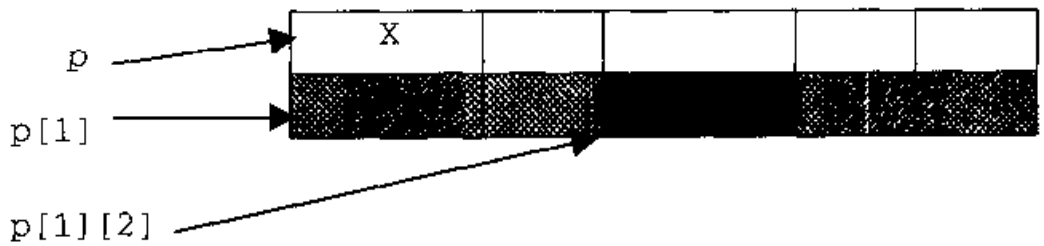
明白了多维数组的概念后，我们就可以理解指向数组的指针[1]。数组指针和多维数组之间同样有密切的关系。例如：

```
int (*p)[5];
```

p 是指针，它指向由 5 个 int 元素组成的数组。

接着，让我们分析一下 p[1][2]代表什么：

很简单，首先 p[1]代表一个 int 数组，如果 p 指向的地址值是 X，那 p[1]就指向（X+20），然后，既然 p[1]的类型是 int 数组，那 p[1][2]就代表一个 int，这个 int 的地址值是（(X+20) + 2*4），即（X+28）。



也就是说，p 和上面我们定义的二维数组 array 用法完全一样，p[m][n]和 array[m][n]都是指偏移量为（m*20+n*4）的那个 int。

事实上，在函数的参数里面，我们完全可以用指针去表示数组，例如：

```
int fun(int array[]);
```

可以写成：

```
int fun(int *array);

int fun(int array[][5]);
```

可以写成：

```
int fun(int (*array)[5]);
```

大家不难分析出，无论写成数组的形式，还是写成指针的形式，意义都是一样的，

运算结果也完全相同。

最后，我们看一下指向指针的指针和多维数组的区别，例如对于：

```
char **argv;
```

和

```
char array[i][j];
```

argv 和 array 的类型相同否？

前者其实很常见，例如 main 函数的其中一种写法就是：

```
int main(int argv, char  **argv){...}
```

如果这样写：

```
int main(int argv, char* argv[]){...}
```

可能更直观一些，两者是等价的。从后者来看，argv 是一个指针数组，它的元素是指向字符的指针，因为在函数的参数里面，数组可以用元素的指针类型表示，即 argv 可以看作指向 char*的指针，也就是 char**。

argv[m]的类型是 char*，argv[m][n]的类型是 char。如果 argv 的值是 X，那 argv[m][n]的地址就是(*(X+4*m)+n)。

而对于二维数组 array，array[m]的类型是有 j 个 char 的数组，array[m][n]的类型是 char。如果 array 的地址值是 X，那 array[m][n]的地址就是(X+j*m+n)。

[FN1901]：指向数组的指针称为“数组指针”，注意区别数组指针和指针数组：

```
int (*p)[5];                    /* 数组指针 */
int *p[5];                      /* 指针数组 */
```

Part III

20 词法分析

C 编译器的工作是把 C 代码转换成对应的目标代码让链接程序进一步生成可执行文件又或者库文件。前面说过，这个过程一般要经过好几个步骤，首先是预处理程序对源文件进行预处理（把头文件包含进来、执行宏替换等），然后是编译程序进行一系列的扫描、分析等。无论是在预处理还是编译阶段，词法分析（lexical analysis）都是一个必不可少的过程，编译器[1]通过词法分析确定字符序列由哪些记号（token）组成并收集这些记号，例如：

```
a=b+5;
```

这行语句由以下记号组成：

记号	类别
a	标识符
=	操作符
b	标识符
+	操作符
5	常数
;	分隔符

你可以用空格明确指示编译器进行词法分析，例如你这样写：

```
a = b + 5 ;
```

否则编译器按照“最大匹配”原则进行分析，意思就是编译器在遇到“空白”（例如空格、回车、制表符等）之前，以能够取得的、有意义的、最长的字符串作为记号，举例如下：

```
iCount++;
```

编译器从“i”开始分析，一直到第一次碰到“+”，这时，由于对前面收集的“iCount”的判断是标识符，现在来了一个“+”，但 C 语言不允许“+”出现在标识符里，于是编

译器就认为这个记号已经结束，它就是“iCount”，类别是标识符。

编译器接着分析，它对第一个“+”的初步判断是操作符，但它仍然继续收集，这时遇到第二个“+”，由于C语言有一个操作符是“++”，所以编译器认为记号还是属于操作符，然后继续取字符，发现“;”，“;”属于分隔符，表示一个语句的结束。因为记号“++;”在C语言里没有意义，所以编译器下判断说这次收集到的记号是“++”，类别是操作符……

正是由于“最大匹配”原则，编译器没有把上面的语句分析为：

```
iCount
+
+
;
```

而是分析成：

```
iCount
++
;
```

如果我们没有良好的编程习惯，就会由于这个词法分析特点招惹小麻烦，例如：

```
int a,b,c;
c=a+++++b;                    /* Error */
```

其实这位程序员的意思是：

```
c = (a++) + (++b);
```

但他过分简洁地书写，于是编译器依照“最大匹配原则”的词法分析结果是：

c	=	a	++	++	+	b	;

随后编译器进行语法、语义分析，得出的结论是以为程序员想这样做：

```
c = ((a++)++) + b;
```

它马上报告错误，因为“a++”的返回值不是左值（lvalue），不能进行自增运算 。

又例如：

```
int a, b, *c;
a=b/*c;
```

写代码的人其实是这样的意思：

```
a = b / (*c) ;
```

但预处理程序的词法分析是：

a	=	b	/*	c	;

“/*”是注释的开始，于是预处理把从“/”开始的内容全部注释掉，直到碰见记号“*/”为止，这显然不是我们想要的结果。

了解词法分析的特点可以帮助我们判断某些问题，例如：

```
#define  SIZE  1000
...
int nSIZE = 512;
...
```

上面一开始就用宏指令把“SIZE”替换为“1000”，你说，随后的变量“nSIZE”会不会被预处理程序替换成“n1000”呢？

不会的。因为预处理程序执行宏替换的单位是记号，不是字符。变量的名字虽然含有“SIZE”，但预处理程序的词法分析表示，在“int”后面的记号是“nSIZE”，这个记号和宏指令定义的记号“SIZE”不匹配，所以不会执行替换。

当你熟悉了词法分析后，写得再刁钻的程序也难不倒你：

```
int
main   (int
argc          ,char
*argv
[       ]
```

```
){int     a=
7
;a
+=6
;
return
0
;}
```

上面的代码虽然太夸张，但却是完全正确的，大家不妨自己分析一下。

[FN2001]：这里和下文所讲的编译器包括预处理程序和编译程序，因为它们都要进行词法分析。

21 注释

C90 只有一种注释风格，就是使用“/*”和“*/”。早在预处理阶段，C 预处理器就会剔除所有出现在“/*”和“*/”之间的内容（包括这对记号本身），所以在接下来传送到 C 编译器的代码文件里是不会出现任何注释内容的。

但这么简单的注释风格有时也会令人头疼。例如，你可能想，什么是注释呢？注释就是写什么东西都无所谓，反正编译器不会看到这些东西。很不幸，C 预处理器一般都不支持嵌套注释，如果你这样写：

```
/* something /* something else */ */
```

是行不通的。因为预处理器在碰到第一个“*/”时就认为注释已经结束！于是它把到这里为止的注释移除，可想而知，结果留下孤零零一个“*/”——编译程序又报错了。

这样的处理常常令我们很不舒服，例如有时我们调试代码，需要把连续几行代码先注释掉观察结果，做法自然是这样写：

```
语句 1;
    语句 2;
    /*
        语句 3;
    */
    语句 4;
语句 5;
```

但随着调试的深入，可能我们想连语句 2、语句 4 都注释掉：

```
语句 1;
/*
    语句 2;
```

```
    /*
        语句 3;
    */
    语句 4;
*/
语句 5;
```

多加一对注释符不就解决了？

——中招了。看，你用了嵌套注释。

为了解决这个问题[1]，C++引入另一种风格的注释：使用“//”。预处理器会删除从“//”开始到后面碰到的第一个换行符之间的所有内容。这样就没有嵌套注释的问题了，只不过书写有点繁琐，例如：

```
语句 1;
// 语句 2;
// 语句 3;
// 语句 4;
语句 5;
```

你喜欢注释哪一行都可以，随时可以撤消注释，干脆利落，绝对没有副作用，当然，前提是你一行只写一个语句。

资深程序员一般都更乐意使用 C++风格的注释，我也如此建议。不过，你会问，C 预处理器可不认识“//”啊？

其实多数 C 预处理器都“暗中”支持“//”，除非你使用“严格符合 C90”之类的编译选项，否则 C 预处理器是不会阻挠你的。

而且，有个好消息：

C99 已经正式采纳 C++的注释风格，在今后，你可以光明正大地使用“//”。当然，原来的老式风格仍然可以继续使用。

不过，大家最好不要“一只脚踏两只船”，像这样就不好：

```
// /* 语句1;
语句2;
*/
语句3;
```

原先用“/*”和“*/”注释掉语句1和语句2，现在不小心又加上“//”，出错了。因为预处理器一看见“//”就不管三七二十一把“// /* 语句1;”全部删除掉，结果又留下一个“*/”没有处理。

结束本节之前，再讨论一个有趣的问题。

有些敏感的读者已经坐不住了：那万一我想在字符串里包含“//”之类的注释符怎么办？譬如我要在屏幕上显示这么一行：

```
We have a // in this line.
```

能够这样写吗：

```
printf("We have a // in this line.\n");
```

答案是可以的

只要记住三条原则：

#1 字符串里面的注释符不起注释作用；

#2 注释符里的双引号不起标识字符串的作用；

#3 如果情况混乱则从头找起。

例如：

```
printf(" We have /* and */ here. ");
```

根据原则#1，最后打印出来的字符串不会缺少“/* and */”这部分。

```
// "This line is a comment. "
```

由原则#2，可以肯定这一行会被注释掉。

```
/*  "  */  "*/
```

你可能这样想，第一个“*/”在双引号里面，噢，它不起注释作用，所以这一行会被完全注释掉。

不是的。在这么“恶劣”的环境下，我们要找出最先出现的符号是注释符还是双引号，在这个例子里，注释符最先带头，所以根据原则#2，双引号不起任何作用，当预处理器碰到第一个“*/”时就认为注释结束，预处理的结果是上面这一行语句最后剩下：

```
"  */
```

[FN2101]：C/C++程序员可以使用预处理指令实现嵌套注释：

```
语句 1;
    #if 0
        语句 2;
            #if 0
                语句 3;
            #endif
        语句 4;
    #endif
语句 5;
```

因为预处理器在分析预处理指令时不仅进行词法分析，还进行语法分析，这就比使用“/*”和“*/”高级，后者仅仅涉及词法分析。

22 优先级与运算顺序

C 语言的操作符虽然不算很多，但如果有好几个操作符同时出现在同一表达式里面并且没有加括号帮助理解，有时会令人颇为头疼。一不小心就会掉进陷阱半天出不来，甚至会莫名其妙：我的程序怎么啦？

譬如下面就是一种常见的错误：

```
#define  BUSY  0x8000
...
    while (readPort(portNum) && BUSY != 0)
        ;
    doSomething();
...
```

上面代码的本意很简单，读某个端口 portNum，要是返回值的最高位是“1”就表示系统忙，需要等待，否则就可以进行某种操作 doSomething()。

但是，“!=”比“&&”的优先级高，所以上面的循环语句相当于：

```
while (readPort(portNum) && (BUSY != 0))
    ;
```

看看后结果，BUSY 就肯定是“!-”0 的，于是“&&”右面的表达式返回“1”，而 readPort(portNum)的返回值如果最低位不是“0”的话，“&&”运算的结果就是“非零”——程序进入死循环跳不出来。

正确的写法应该是：

```
while ((readPort(portNum) && BUSY) != 0)
```

在实际的编程中，类似的错误数量非常多。所以，搞清楚 C 语言的操作符优先级是很重要的，这里给出下面的表格：

操作符	结合类型
() [] -> .	左结合
! ~ ++ -- - (type) * & sizeof	右结合
* / %	左结合
+ -	左结合
<< >>	左结合
< <= > >=	左结合
== !=	左结合
&	左结合
^	左结合
|	左结合
&&	左结合
||	左结合
?:	右结合
赋值符（ = += -= *= /= %= 等）	左结合
	左结合

依照上表，很容易判断表达式的意思，例如：

```
*p++ = 5;                    // 假设 p 是 int 指针
```

上面这条语句的操作是：

`*(p++) = 5;` 即先对 p 指向的 int 赋值 5，然后 p 自增[1]。

以下的式子相信大家很熟悉：

```
while ( ch = getc(fp_in) != EOF )
```

```
    putc(ch, fp_out);
```

小心，“!=”比“=”优先级高。这样写是错的，应该写成：

```
while ( (ch = getc(fp_in)) != EOF )
    putc(ch, fp_out);
```

如果你实在不愿意记具体的优先级究竟谁高谁低，请尽量使用括号，把你想要的操作明确表示出来。

下面我们再说一下运算的顺序。

例如：

```
a = b + c + d;
```

初学者可能这样认为：

编译器一定会首先安排 b 和 c 相加，然后再把得出的和与 d 相加，最后赋值给 a，即使说：

```
a = (b + c) + d;
```

但是实际的情况是 C 标准并没有对此作出明确的规定，在某些平台又或者某些情形下，可能编译器是这样实现的：

```
a = b + (c + d);
```

类似的例子还有：

```
a = f() + g();
```

编译器并不保证 f()一定会在 g()之前被求值。

再有：[2]

```
int i = 0;
while (i < 10)
```

```
arrayTarget[i] = arraySource[i++];
```

代码写得很简洁，作者其实想：

```
while (i < 10)
{
    arrayTarget[i] = arraySource[i];
    ++i;
}
```

虽然 i++返回自增前的 i 值，但问题是某些平台的编译器并不能保证在 i 自增之前执行取址操作：

```
arrayTarget[i]
```

所以最好还是老老实实写：

```
int i = 0;
while (i < 10)
{
    arrayTarget[i] = arraySource[i];
    ++i;
}
```

总之，不要对运算的顺序作过多的假设，除非像这样极度简单的式子：

```
a = b * c + d;          /* 乘法一定会在加法之前进行运算 */
```

[FN2201]：注意，p 是 int 指针，32 位平台上 p++的结果是 p 的数值增加 4。

[FN2202]：下面的这个例子出自[Koenig, 1989]。

23 友好的 typedef

C 语言允许我们自己定义新的数据类型，其语法很简单，例如：

```
typedef int INT32;
INT32 a;                /* 相当于 int a; */
INT32 *p;               /* 相当于 int *p; */
```

可以看出，如果想用一个新的名字 XXX 代表一种数据类型，只要声明一个这种数据类型的变量，变量名是 XXX，并在语句的最前面加上“typedef”关键字即可。

例如：

我们想建立 String 数据类型，它其实是 char*，

第一步，声明一个 char*变量 String；

```
char *String;
```

第二步，加上 typedef。

```
typedef char *String;
```

这样，从现在开始我们就可以用 String 表示 char*，例如：

```
String fileName;
int main(int argc, String argv[]){}
```

在使用 typedef 的过程中要注意两点：

#1 typedef 跟变量一样有可视范围，并且内层的可以覆盖外层的。

在不同的函数内部，可以使用不同的名字表示同一种数据类型，或者是使用相同的名字表示不同的数据类型。例如：

```
C01   /* Example code 23-01, file name: 2301.c */
C02
C03   int main(void)
```

```
C04    {
C05        typedef int INT32;
C06        INT32 a;
C07        return 0;
C08    }
C09
C10    void fun()
C11    {
C12        typedef long INT32;
C13        INT32 b;
C14    }
```

这个例子里面，main 和 fun 各自定义自己的新数据类型 INT32，但具体所指并不一样，main 的 INT32 是 int，fun 的 INT32 是 long，但它们在自己的作用范围里都能独立起作用，互不干扰。

又例如：

```
C01    /* Example code 23-02, file name: 2302.c */
C02
C03    typedef int INT32;
C04    INT32 a;
C05
C06    int main(void)
C07    {
C08        typedef long INT32;
C09        INT32 b;
C10        return 0;
```

```
C11    }
```

这个例子也是在不同的区块里面定义不同的 INT32，一个是在本编译单元（即 C 文件 2302.c）的开始，所有函数的外面；另一个是在 main 里面。它们各自在不同的范围工作。

#2 （在同一作用范围内）不能用相同的名字定义不同的数据类型。

例如：

```
C01    /* Example code 23-03, file name: 2303.c */
C02
C03    int main(void)
C04    {
C05        typedef int INT32;
C06        typedef long INT32;        /* Error */
C07        return 0;
C08    }
```

```
C01    /* Example code 23-04, file name: 2304.c */
C02
C03    int main(void)
C04    {
C05        typedef int INT32;
C06        typedef int INT32;        /* Error */
C07        return 0;
C08    }
```

可以看到，即使一模一样的 typedef 也不能重复出现[1]。

当然，用不同名字表示相同数据类型是可以的：

```
C01   /* Example code 23-05, file name: 2305.c */
C02
C03   int main(void)
C04   {
C05       typedef int INT32;
C06       typedef int INT_4BYTE;       /* OK */
C07       INT32 a;
C08       INT_4BYTE b;
C09       return 0;
C10   }
```

而且可以把新定义的数据类型当作固有数据类型使用：

```
C01   /* Example code 23-06, file name: 2306.c */
C02
C03   int main(void)
C04   {
C05       typedef int INT32;
C06       typedef INT32 INT_4BYTE;       /* OK */
C07       INT32 a;
C08       INT_4BYTE b;
C09       return 0;
C10   }
```

typedef 的作用范围最大只能是它所在的编译单元，不同 C 文件里面的 typedef 互相独立，完全没有任何限制。

有些读者可能会问：预处理指令“#define”也能完成类似的功能，为什么还要多此一举搞出个 typedef 呢？

预处理指令执行的仅仅是低级的记号替换，某些时候就会暴露出问题：

```
#define char* String;
String inputFileName;
```

预处理程序把“#define”指令后面的文本中所有的记号“String”替换成“char*”，所以上面的代码可以工作。但如果这样写：

```
#define char* String;
String inputFileName, outputFileName;
```

经过替换，上面的语句变成：

```
char* inputFileName, outputFileName;
```

很明显，这时 outputFileName 的类型不是 char*，而是 char！[2]

这就是#define 的局限。

typedef 就没有这个问题，因为 typedef 是由编译器进行语法分析的，用它定义的类型对声明（或定义）语句中的每一个变量都起作用：

```
typedef char* String;
String inputFileName, outputFileName;
```

相当于写：

```
char *inputFileName, *outputFileName;
```

一旦我们掌握 typedef 这个利器，就应该好好发挥它的威力。例如下面非常复杂的写法[3]：

```
(*(void (*)())0x12345678)();
```

完全可以用 typedef 把它改写得友善一点：

```
typedef void (*PF)();
```

```
(*(PF)0x12345678)();
```

不过，typedef 也有一点“局限”，用它定义的类型不能“组合使用”，例如：

```
typedef int INT32;
unsigned INT32 a;                    /* Error */
```

别以为上面的声明会被编译器看作是：

```
unsigned int a;
```

不是的，编译器会拒绝这样的“组合”。

相反，使用#define 却没有这个问题：

```
#define int INT32
usigned INT32 a;                     /* OK */
```

并且，在某些时候，使用 typedef 一定要小心，例如：

```
typedef  char *String;
const String s;
```

上面定义的 s 究竟是：

```
const char* s;              //s 是一个指向 const char 的指针
```

还是：

```
char *const s;              //s 是一个指向 char 的 const 指针
```

答案往往会让粗心的人吓一跳，正确的解释是后者。因为当用 typedef 定义了一种新的类型 String 之后，const 修饰的对象就是 String，而 String 本身是指针，于是 const String 的意思就是 String（某种指针）的值是常量，所以最后 s 就被理解为指向 char 的 const 指针。[4]

[FN2301]：C++允许完全相同的 typedef 表达式多次出现，即：

```
int main(void)
{
    typedef int INT32;
    typedef int INT32;          /* OK in C++ */
```

```
}
```

[FN2302]：别忘了，“*”是右结合的：

```
char* inputFileName, outputFileName;
```

相当于：

```
char (*inputFileName), outputFileName;
```

[FN2303]：这个例子最早出现在第 17 小节。

[FN2304]：关于 const 的用法请参阅后面的章节。

24 C-V 限定词

K&R C 中并没有 C-V 限定词的概念，它源于 C++。在 C90 标准的制定过程中，C-V 限定词逐渐被广泛接受，最终也成为 C90 的一部分。

C-V 限定词是指 **const** 和 **volatile** 这两个修饰符，它们的用法很简单，首先讨论 const，例如：

```
const int a = 1;
```

这样，由于变量 a 有了 const 修饰，则 a 就不能被修改。编译器如果检测到企图修改 a 的代码就会报错：

```
a++;                    /* Error */
```

在同一行语句中，不同位置的 const 可以修饰不同的对象：

```
int a = 0;
const int *p = &a;
(*p) = 1;                    /* Error */
```

上面的 const 修饰 int 指针 p 指向的对象，所以我们不能修改（*p）。

如果这样写：

```
int a = 0, b = 1;
int * const p = &a;
(*p) = 1;                  /* OK */
p = &b;                    /* Error */
```

这一次 const 修饰的是指针 p 本身的值，也就是说 p 的值不能被改变，但 p 指向的 int 却可以修改。

如果这样：

```
int a = 0, b = 1;
const int * const p = &a;
```

```
(*p) = 1;                /* Error */
p = &b;                  /* Error */
```

这里用了两个 const，一个修饰指针 p，另一个修饰 p 指向的 int，所以无论修改 p，或者修改（*p）都是错误的，编译器不会允许这样的代码通过编译。

通过使用 const，我们可以指望编译器为我们的代码进行有益的检查，最突出的例子就是用 const 修饰函数的参数。

当我们把一个指针作为参数传给某个函数时，我们可能只是希望函数能够引用到指针指向的内容但却绝对不允许函数改变它，这个时候我们就要出动 const：

```
int memcmp(const void *s1, const void *s2, size_t n);
```

这是 C 标准库的一个函数，其功能是比较由 s1 和 s2 指向的内容是否相同。既然是比较，我们就要求函数只是读取指针指向的内容，函数根本不需要（也不应该）改变内容本身。为了保证函数不能修改指针指向的内容，我们用了 const 修饰参数。万一写这个函数代码的程序员没有遵守规则，又或者他一时疏忽大意，编译器就会报告错误。

不过，const 在 C 里面的用法和在 C++里面仍然有不少差别：

#1 C++能够把（已用常量赋值的）const 变量看作编译期常数，C 没有这种功能。

例如：

```
const int BUFSIZE = 1024;
char buf[BUFSIZE];       /* Valid in C++ but illegal in C */
```

所以，如果 C 程序员想定义编译期常数，还得乖乖地依靠预处理指令：

```
#define BUFSIZE 1024
char buf[BUFSIZE];       /* OK both in C and C++ */
```

#2 C++默认 const 变量的链接性质是内部的，而 C 则相反，默认是外部的。

例如：

```
const int a = 0;
int main(void){ }
```

上面的变量 a 由于是在函数的外部定义并且没有 static 修饰，所以在 C 语言里面

毫无疑问应该是外部链接的，即其他文件的代码能够访问到它。

但在C++中，变量a默认是内部链接的，除非你显式加上extern修饰词，否则，其他文件是看不到const变量a的。

#3 C只能允许用常量初始化const外部变量，C++没有这种限制。

例如：

```
int f(void);
const int a = f();          /* Valid in C++ but illegal in C */
int main(void){ }
```

上例中，虽然f()返回int值，但C编译器是不允许这样初始化const变量的，要初始化a，只能用常数，例如：

```
const int a = 2;
```

无论C和C++在这方面有什么差别，可以预期的是，很多编译器（包括C和C++编译器）都会把const外部变量放到只读数据区，这样就能进一步确保const变量不被修改。

例如：

```
C01     /* Example code 24-01a, file name: 2401a.c */
C02
C03     const int a = 5;
```

```
$gcc  -S  2401a.c
$less  2401a.s
```

```
A01     .globl a
A02         .section  .rodata
A03         .align 4
A04         .type  a,@object
```

```
A05         .size  a,4
A06     a:
A07         .long 5
```

留意A02，这一行指示编译器把a放入只读数据区。于是，C语言中const外部变量的默认链接性质会引起一点小问题：

```
C01     /* Example code 24-01, file name: 2401.c */
C02
C03     extern int a;
C04     int main(void)
C05     {
C06         a++;                        /* 修改const变量！ */
C07         return 0;
C08     }
```

看看2401.c，由于C03没有把a声明为const，所以编译能够顺利通过。链接程序呢，不要指望它，它仅仅为你安排好全局变量a的地址，至于a是否只读而你又是否进行了修改，它一概不管。以下命令成功执行：

```
$gcc  2401.c  2401a.c
```

但是，当执行程序时：

```
$./a.out
Segmentation fault
$
```

系统报告说发生了错误。因为该代码企图修改只读数据区的数据！编译完全没问题，但执行时出错。

为什么C++中const外部变量默认是内部链接呢？因为C++设计思想的其中一点

就是：错误应该尽可能在编译期被检测出来。编译器完全可以监视同一文件内的 const 变量是否被本文件中的代码非法修改[1]，除非（固执地）把 const 变量显式声明为 extern。

最后我们说一下 volatile 的用法。用 volatile 修饰一个变量的意思是告诉编译器这个变量可能会被外部事件修改（例如硬件本身的状态变化），于是编译器在优化代码时就会顾及这一点。

譬如写：

```
volatile unsigned *port = (volatile unsigned*)0x12345678;
unsigned valueOne, valueTwo;
valueOne = *port;              //第 1 次读端口
wait(1000);                    //等待 1000 毫秒
valueTwo = *port;              //再次读端口
```

上面的代码首先把端口值 0x12345678 赋给指针 port，这样，我们读取 port 指向的 32 位值就相当于读端口[2]。如果不用 volatile 修饰 port，编译器看到我们两次读取同一个指针指向的内容而指针及它指向的存储区又没有被修改过（从代码上看的确如此），它就会优化我们的代码，譬如把第一次读端口得到的值保存在某个寄存器，当再次读端口的时候就直接把寄存器的值作为结果赋给 valueTwo。但由于端口值会随时间变化，所以 valueTwo 的值非常可能不是我们想要的。

在这种情况下，volatile 的作用就是通知编译器在优化代码时不要擅自作出假定而导致错误的结果。

const 与 volatile 既可以单独使用，也可以一起使用：

```
const volatile char* port = (const volatile char*)0x01F7;
```

这样，port 就成为一个只读端口。

[FN2401]：因为 const 外部变量默认是内部链接的，所以访问 const 变量的代码必须和 const 变量同在一个文件里面，这样，编译器就可以检测出非法修改。

[FN2402]：不像 IA，某些计算机架构的端口和内存是共享同一个地址空间的，也就是说，读写端口和读写内存没什么不同，用的是相同的指令，只不过地址值不同罢了。这里举的例子也是基于这样的假设。

25 字符串

我们经常和字符串打交道，但是，我们真的就对它了如指掌吗？看看：

```
C01    /* Example code 25-01, file name: 2501.c */
C02
C03    int main(void)
C04    {
C05        char *p = "ABCDE";
C06        *p = 'Z';          /* 这一句导致运行期错误 */
C07        return 0;
C08    }
```

这个程序顺利通过编译：

```
$gcc  2501.c
```

但是一执行就会：

```
$./a.out
Segmentation fault
$
```

我们把字符指针 p 指向字符串“ABCDE”（C05），然后通过修改*p（C06）来改变字符串的第一个字母，有什么问题啊？

——别忘记，字符串是常量的一种，不能被改变，所以例子中的字符串会被编译器安排到只读数据区，我们可以通过 p 读取字符串但绝对不能修改字符串，否则就会发生运行期错误。

严格地说，C05 应该这样写：

```
const char *p = "ABCDE";
```

但由于以往大量的 C 代码都没有这样写，所以编译器也接受 C05 的写法，不过就得自己去保证代码不会修改字符串，否则编译器是“爱莫能助”。

如果这样写：

```
C01    /* Example code 25-02, file name: 2502.c */
C02
C03    int main(void)
C04    {
C05        char array[] = "ABCDE";
C06        array[0] = 'Z';          /* OK */
C07        return 0;
C08    }
```

就完全没问题。

这一次，字符串同样是常量，但我们只是用它来初始化字符数组 array。编译器在 main 函数的栈里面分配空间给 array，然后把字符串的所有字符复制到 array 数组。到此，字符数组的初始化工作结束。C06 修改的不是字符串而是字符数组 array 的元素 array[0]。

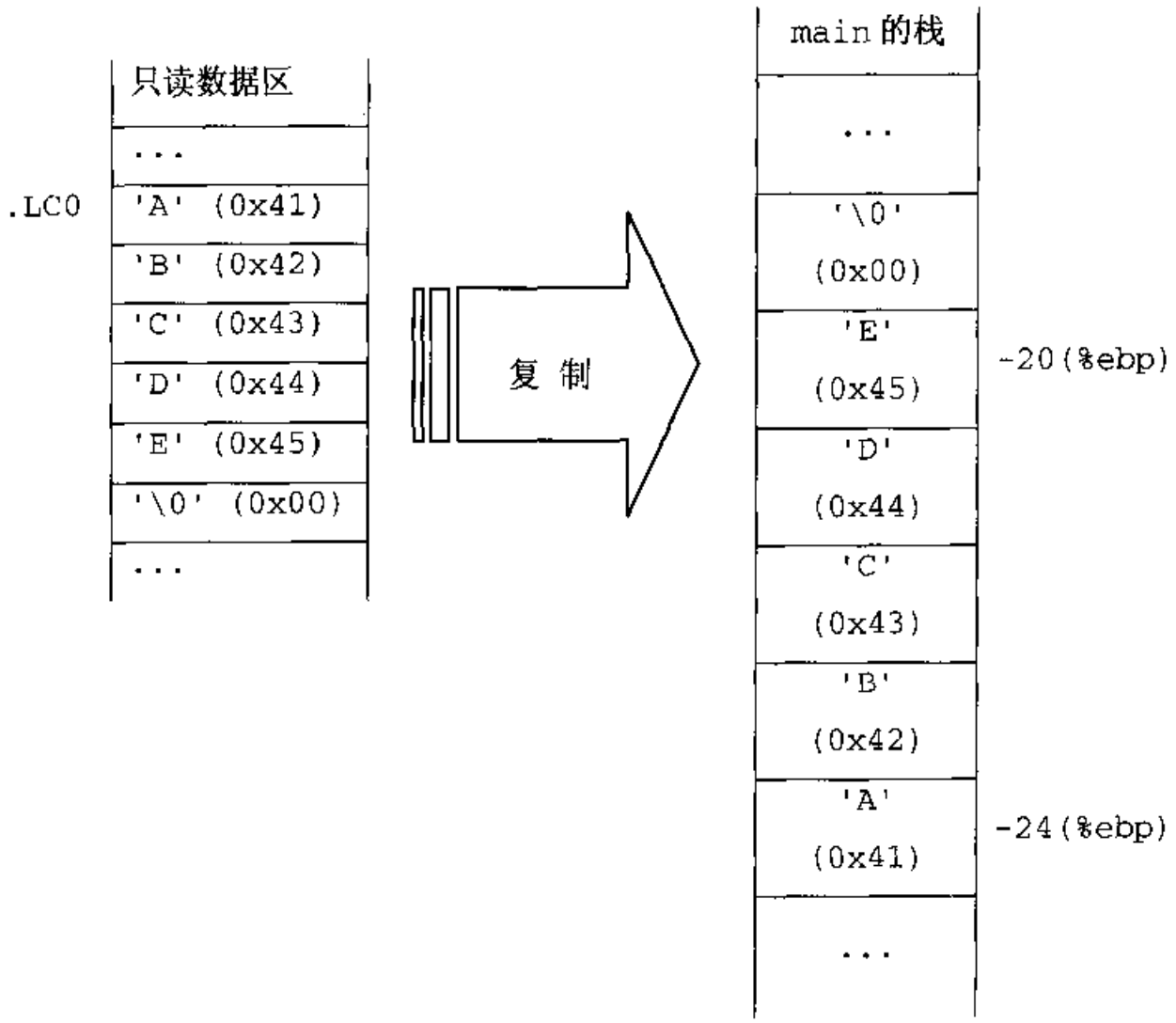
以下是汇编代码：

```
A01        .section  .rodata
A02    .LC0:
A03        .string   "ABCDE"
A04    .globl main
A05        main:
A06        ...
A07        movl   .LC0, %eax
```

```
A08         movl   %eax, -24(%ebp)
A09         movzwl .LC0+4, %eax
A10         movw   %ax, -20(%ebp)
A11         movb   $90, -24(%ebp)
A12         ...
```

A01 的“.rodata”明确指出字符串“ABCDE”仍然放在只读数据区。

A07 把从地址“.LC0”开始的 4 个字节（即“A”、“B”、“C”、“D”4 个字符）复制到 eax，再由 eax 复制到栈（A08），接着，剩下的两个字节也是用同样的方法复制到数组（A09、A10）。大家可以看到 A11 修改的是栈里面的 array 元素，而不是只读数据区的字符串。[1]



看了两个例子，相信大家对字符串的认识已经有一个基本的框架，现在我们再进一步讨论，是不是所有的字符数组都要通过放在只读数据区里的字符串来初始化。看看下面的例子：

```
C01    /* Example code 25-03, file name: 2503.c */
C02
C03    char array[] = "ABCDE";
C04    int main(void)
C05    {
C06        array[0] = 'Z';                    /* OK */
C07        return 0;
C08    }
```

这次我们定义了全局字符数组 array，并用字符串“ABCDE”初始化数组，看看汇编代码有没有变化：

```
$gcc  -S  2503.c
$less  2503.s
```

```
A01        .globl array
A02            .data
A03            .type  array,@object
A04            .size  array,6
A05        array:
A06            .string   "ABCDE"
A07        .globl main
A08            .type  main,@function
A09        main:
```

A10	...
A11	**movb $90, array**
A12	...

可以看到，编译器并没有在只读数据区额外放置字符串常量（因为已经不需要这样做），而是直接安排初始化字符（A06），当然，修改数组的任何元素也是没有问题的。

同理读者可以自行分析加上 static 修饰的情况，例如：

```
static char array[] = "ABCDE";
int main(void){}
```

和

```
int main(void)
{
    static char array[] = "ABCDE";
}
```

中的字符串是否被安排在只读数据区。

[FN2501]：IA 是属于 little-endian 的。

26 void 表示什么

关键字 void 同样是 C90 新增加的，它的用法涉及几个方面。

还记得我们讲过的函数原型吗？

函数原型为编译器提供依据检查函数的参数是否合乎要求。但由于大量 K&R C 代码只是像这样声明函数：

```
double fun();
```

所以基于兼容性的考虑，C90 一看到上面的声明，就不对函数的参数进行匹配检查（因为根本没办法检查）。

那么，如果一个函数没有任何参数，我们又想通过函数原型让编译器进行参数检查，该怎么写呢？

这时，我们需要使用 void 明确指出：

```
double fun(void);
```

这样编译器就知道 fun 是没有参数的，一旦它发现类似：

```
fun(50);
```

的调用操作，就能够马上报错。

现在，你应该明白，在 C 语言里面：

```
int fun();
```

和

```
int fun(void);
```

具有完全不同的涵义。[1]

C90 还增加了一种指针：void*。void 指针指向的对象是不确定的，可能是 int，也可能是一个 struct，总之，void*可以指向任何对象（除了函数）。

void 指针具有以下两个主要特性：

#1 它可以和其他任何指针（函数指针除外）相互转化而不需类型强制转换。

例如：

```
int *pi;
void *pv;
...
pi = pv;          /* OK */
pv = pi;          /* OK */
```

#2 不能对它执行取值和下标操作。

这是很容易理解的，因为void*究竟指向什么完全不知道，像下面的操作根本不可能有意义：

```
void *pv;
...
*pv = 5;          /* Error! (*pv)究竟是什么，占多少个字节？ */
pv[0] = 3;        /* Error! 理由同上 */
```

那void*又有什么用处？

在K&R C时代，“通用指针”这个角色是由char*扮演的，因为一个char占一个字节，这和存储器以字节为最小单位的概念相吻合。但是char指针和其他类型的指针互相赋值需要强制转换，因为它们指向的对象不相同，例如：

```
int *pi;
char *pc;
...
pi = (int*)pc;                  /* 必须进行强制转换 */
pc = (char*)pi;                 /* 同上 */
```

于是某些函数用起来就有些不方便了。譬如函数：

```
memcmp(s1, s2, n)
char *s1, *s2
{ ... }
```
[2]

顾名思义，memcmp 的功能是比较从内存地址 s1 和 s2 开始、最多 n 个字节的内容是否相同。很明显，我们调用函数的参数不一定都是 char*，例如可能想比较两个 struct 变量是否相同，这样就必须把 struct 指针强制转换成 char 指针：

```
struct s *ps1, *ps2;
memcmp((char*)ps1, (char*)ps2, sizeof(struct s));
```

大家有没有发觉这样写比较麻烦？本来意思是比较两块内存区域是否相同，根本不需要理会作为参数的指针指向什么类型的变量，它可以指向 int，也可以指向一个结构变量，反正它是一个指向对象（函数除外）的指针就行，这时，void*便大显身手了：

```
int memcmp(const void *s1, const void *s2, size_t n);
```

像上面要比较两个 struct 指针指向的区域，这样调用函数就显得非常简洁：

```
memcmp(ps1, ps2, sizeof(struct s));
```

大家对类似下面的代码应该相当熟悉：

```
int *pi = (int*)malloc(100 * sizeof(int));
```

上面调用标准库函数 malloc()向系统申请 100 个 int 的空间。

还记得（那些十几年都不更新的）C 语言教材是怎样解释的吗：

——因为 malloc 返回的是 char*，所以要强制转换成需要的指针类型。

其实……看看 C90 的函数库说明吧：

```
void *malloc(size_t size);
```

标准写明 malloc 返回的是 void* ！我们完全可以这样写：

```
int *pi = malloc(100 * sizeof(int));        /* OK! */
```

记住，在 C 语言里面，void 指针可以自动转换为其他任何类型的指针（函数指针除外）。

不过，C++的设计者却认为这不是一个好主意，于是他们坚持强制转换是必须的，因此，在 C++中我们必须写：

```
int *pi = (int*)malloc(100 * sizeof(int));
```

如果要让我们的C代码能够同时被C/C++编译器接受，当涉及把void指针的值赋给其他类型的指针时[3]，（不怕麻烦地）加上类型强制转换是惟一的办法。

最后，我们看一看在C/C++中NULL的实现。大家都知道，NULL表示空指针，即指针的值为常数0，通常，C编译器这样实现NULL：

```
#define NULL (void*)0          //NULL in C style
```

由于void指针可以不需要强制转换就赋值给任何其他指针（函数指针除外）[4]，所以，我们用NULL很方便：

```
int *pi = malloc(100 * sizeof(int));
if (pi == NULL)
    ...
else
    ...
```

但在C++就有点麻烦，C++对类型的检查非常严格，任何指针（包括void指针）都不能不通过强制转换就直接赋值给其他指针。于是，究竟应该怎样定义NULL呢？因为无论把NULL定义为何种指针类型，在使用中都非常不方便：

譬如说，如果我们在C++中这样定义NULL：

```
#define NULL (void*)0
```

那就得这样写代码：

```
int *pi = (int*)NULL;
char *pc = (char*)NULL;
```

这意味着数以百万计的C代码无法通过C++编译器的编译，因为C程序员绝对不会这样写代码。于是，C++的设计者权衡利弊后，接受了下面的实现[5]：

```
#define NULL 0                 //NULL in C++ style
```

所以，无论如何，我们这样写绝对没问题：

```
int *pi = NULL;                //OK in C/C++ now
```

[FN2601]：对 C++来说，上面两种声明都表示同一个意思，就是函数没有参数，即是说，（参数列表里面的）void 可写可不写。

[FN2602]：这是 K&R C 风格的代码，因为在 void 引入之前，标准库都是用这种风格的代码写的。

[FN2603]：注意，在 C++中，如果把其他类型指针的值赋给 void 指针，则不需要强制转换，这是和 C 一样的。例如：

```
int *pi;
void *pv;
...
pv = pi;                          //OK in C/C++!
```

[FN2604]：NULL 是一个特殊的 void 指针，因为它的数值为常数 0，所以它能够赋值给函数指针。例如：

```
int (*pf)(void);
pf = NULL;                    // OK
```

但除了 NULL 外的其他 void 指针是不能直接赋给函数指针的：

```
int (*pf)(void);
pf = (void*)1234;             //Error
```

[FN2605]：这个实现基于 C/C++语言的一个语法规定，就是常数 0 能够无须任何强制转换就赋给任何指针，例如：

```
int *pi;
int (*pf)(void);
pi = 0;              //OK in C/C++
pf = 0;              //OK in C/C++
```

27 #pragma 与 _Pragma

C90 为预处理指令家族带来一位新成员：#pragma。一般情况下，大家很少见到它。

#pragma 的作用是为特定的编译器提供特定的编译指示，这些指示是具体针对某一种（或某一些）编译器的，其他编译器可能不知道该指示的含义又或者对该指示有不同的理解，也就是说，#pragma 的实现是与具体平台相关的。

为了让大家了解#pragma 的用法，这里暂时以 HP C Compiler 为例子[1]。HP C 编译器主要运行在 HP-UX 平台上，它的一般用法和 gcc 大致相同，例如要编译程序：

```
$cc  example.c
```

如果我们明确指示编译器优化代码，可以这样编译：

```
$cc  +On  example.c
```

其中 n 可以是 1、2、3、4，分别代表不同程度的优化，例如：

```
$cc  +O2  example.c
```

这条命令指示 HP C 编译器对整个 example.c 的代码采取第 2 级别的优化编译。

但是，某种情况下我们可能需要特殊对待某一部分的代码，譬如暂停优化，HP C 编译器提供几种途径去实现，其中之一是使用#pragma：

prog.c

```
void f(){...}


#pragma OPTIMIZE OFF
int g(){...}
#pragma OPTIMIZE ON


double h(){...}
```

```
$cc  +O2  prog.c
```

上面的 prog.c 中有 3 个函数，我们想用第 2 级别的优化编译代码整个文件，惟独

函数 g()例外，我们出于某种考虑决定不对它进行任何优化，可以看到，用两条#pragma 指令就能够达到目的。

第一条#pragma 指令指示编译器停止优化代码，于是 g()的代码是没有经过优化的。第二条#pragma 指令通知编译器重新开始代码的优化编译（优化级别仍然是先前命令行给出的 Level 2），所以从 h()开始的代码又都是经过优化的。

这里以代码的优化编译为例简单介绍了#pragma 的用法，读者必须记住：具体的#pragma 指令在不同情况下可能有不同的效果。假设有两家厂商各自推出自己的 C 编译器，可能真会这么巧同时使用相同的#pragma 指令，偏偏它们的实现又不相同，这样编译的代码就可能会出现意想不到的结果。所以，为了保证#pragma 指令能够被正确地解释，我们通常需要利用其他的预处理指令给予配合，例如：

```
...
#ifdef __hpux
    #pragma  FLOAT_TRAPS_ON  _ALL
#endif
...
```

上例中，只有定义“__hpux”宏的 HP C 编译器才会看到#pragma 指令，其他编译器，例如 gcc，是根本不会看到它的，因为 gcc 不会去定义“__hpux”宏，所以早在预处理阶段，#pragma 指令的内容就被预处理程序删掉了。

有读者可能会问：如果万一编译器看到它不认识的#pragma 指令会报错吗？

答案是：不会。

具体到某一条#pragma 指令的涵义不是 C 标准的管辖范围，编译器不能够因为看到不认识的#pragma 指令就说程序有错，惟一的做法是忽略它。

例如：

```
C01   /* Example code 27-01, file name: 2701.c */
C02
```

```
C03   #pragma   UNKNOWN_DIRECTIVE   UNKNOWN_OPTION
C04
C05   int main(void)
C06   {
C07       return 0;
C08   }
```

```
$gcc  2701.c
$./a.out
$
```

尽管 gcc 不认识上面代码中的#pragma 指令，但编译 2701.c 是完全没有问题的。

现在，C99 提供新的关键字“_Pragma”完成类似的功能，例如：

```
#pragma OPTIMIZE OFF
```

在 C99 中可以写成：

```
_Pragma ("OPTIMIZE OFF")          /* 注意：语句后面是没有分号的！ */
```

“_Pragma”比“#pragma”（在设计上）更加合理，因而功能也有所增强。

例如，我们的编译器支持 4 个不同程度的优化等级，如果使用#pragma，则这样写：

```
#pragma OPT_LEVEL n               /*  1 ≤ n ≤ 4  */
```

你会不会觉得每次都要重复写“#pragma ...”很麻烦？如果可以利用宏定义来简化书写就好了：

```
#define OPT_L(x) #pragma OPT_LEVEL x
```

这时我们只须写：

```
OPT_L(3)
```

就相当于写：

```
#pragma OPT_LEVEL 3
```

可惜，在C90里这永远是一个梦想！因为字符“#”在预处理指令中有特殊的用途，跟在它后面的必须是宏的参数名，例如：

```
#define MACRO(x) #x
```

那么，MACRO(example)的替换结果为：

```
"example"
```

可以想象，前面通过#define来定义一个关于#pragma的宏是不可行的。

不过，新的关键字“_Pragma”就很好地解决了问题，由于_Pragma并不带有字符“#”，所以我们可以放心地定义宏：

```
#define OPT_L(x) PRAGMA(OPT_LEVEL x)
#define PRAGMA(x) _Pragma(#x)
```

这时，我们只要写：

```
OPT_L(2)
```

经过预处理后，就成为：

```
_Pragma("OPT_LEVEL 2")
```

即：

```
#pragma OPT_LEVEL 2
```

[FN2701]：刚好手头上有现成的关于HP C Compiler的资料，所以这里就用它作为例子进行讲解。

Part IV

28 声明内部变量

与C程序员相比，C++程序员在声明内部变量[1]时拥有明显的优势：

C代码

```
int f();
int g();
void h(int, int, int);
int main(void)
{
    int i, j, k;
    ...
    i = f();
    ...
    j = g();
    ...
    for (k = 0; k < 10; ++k)
        h(i, j, k);
    ...
    return 0;
}
```

C++代码

```
int f();
int g();
void h(int, int, int);
int main(void)
{
    ...
    int i = f();
    ...
    int j = g();
    ...
    for (int k = 0; k < 10; ++k)
        h(i, j, k);
    ...
    return 0;
}
```

大家能看出上面两者的差别吗？

C90中，内部变量必须在“块（block）”的开始处（即该块的所有代码之前）全部声明完毕，例如上面C代码中main函数的三个内部变量i、j、k都在函数的最前面声明，然后才能使用。但C++代码中的i、j、k却可以在需要用的时候才声明，非常方便。

内部变量在需要用的时候才声明有什么好处呢？

最主要的好处是这样使代码更加容易维护。在那些比较长的函数里面，如果变量都在最前面声明，当我们分析或修改后面的代码时就不得不经常翻回到前面去阅读变量的

声明，翻来翻去，既麻烦又容易出错。

不过，从现在开始，C 程序员不用再羡慕别人。这种与代码混合的变量声明方式正式被 C99 接纳。从此，C 语言的内部变量声明同样变得灵活自然。

特别是 C99 已经允许我们在 for 语句的初始化表达式中声明变量，而且这些变量的生存期只限于 for 语句，以后写 for 循环语句就变得很轻松：

```
C01   /* Example code 28-01, file name: 2801.c */
C02
C03   #include<stdio.h>
C04   int main(void)
C05   {
C06       int i = 9;
C07       for(int i = 0; i < 3; ++i)
C08           printf("Inside for statement, i = %d\n", i);
C09       printf("Outside for statement, i = %d\n",i);
C10       return 0;
C11   }
```

```
$gcc  -std=c99  2801.c
$./a.out
Inside for statement, i = 0;.
Inside for statement, i = 1;
Inside for statement, i = 2;
Outside for statement, i = 9;
$
```

看，我们再也不用担心 i 是否已经被声明或有其他用途。C06 定义了变量 i，它的初值是 9，虽然 C07 又定义了一个变量 i，但这两个 i 是独立工作、互不干扰的。C07

的i仅存在于for语句里面，一旦退出循环，C07的i便不能被访问，所以C09处printf打印的是C06定义的i值。

[FN2801]：注意，本节所说的内部变量包括静态和非静态内部变量。

29 更严格的类型检查

让我们简要回顾一下C90里面的一些隐含规则：

#1 声明中不带类型关键字的变量被默认为是int变量：

```
const a = 0;          /* 相当于写 const int a = 0; */
static b = 3;         /* 相当于写 static int b = 3; */
```

#2 没有声明返回值的函数定义，其返回值类型默认是int：

```
fun(){ }
```

相当于写：

```
int fun(){ }
```

#3 如果没有函数原型，对于传给函数的参数，整型变量默认调整为int压栈，浮点型默认调整为double再压栈，返回值则认为是int。

不过，从现在开始，C99要求变量和函数在使用前必须明确声明，对于变量，编译器要准确知道其类型；对于函数，编译器要清楚知道其原型，否则报错。

所以，上面列举的三种情况不能出现在C99的代码中。

可以想象，由于函数原型是C99强制性要求的，头文件的作用显得更加重要。

不过，对于只带一对空括号的函数声明：

```
int fun();
```

C99仍然和C90一样，不对其参数作任何检查，并且参数仍然按照上面的规则#3作出调整后再压栈：

```
int fun();
int main(void)
{
    fun(5, 3.14);              /* OK both in C90 and C99 */
```

}

现在再说一下 return 语句。大家都知道，return 语句会结束函数的执行并返回到（函数的）调用者。而且 return 语句还可以带有表达式，表达式的求值结果就是函数的实际返回值（如果需要类型转换则先进行类型转换再传递返回值）。例如：

```
int fun(){    return 5.78;    }
```

那么实际上 fun 返回整型值 5，因为函数定义明确指出 fun 返回 int。

目前，C99 比 C90 对 return 语句的规定更明确，C99 指出[1]：

#1 不带表达式的 return 语句不应该出现在返回值不是 void 的函数里面。

即是说，不应该漏掉某些东西：

```
int fun(){    ...    return;    }        /* Error */
```

#2 带有表达式的 return 语句不应该出现在返回值是 void 的函数里面。

即是说，不应该“无中生有”：

```
void fun(){    ...    return 0;    }        /* Error */
```

另外，就像 C++编译器那样，如果 main 函数最后没有 return 语句，C99 编译器会自动加上去。例如写：

```
int main(void){  ...  }
```

C99 编译器编译出来的代码相当于：

```
int main(void){  ...   return 0;  }
```

[FN2901]：由于有大量的旧代码存在，下面讲到的问题，考虑到兼容性，C99 编译器通常只是发出警告而不是报错。

30 _Bool 的加入

C90 没有“布尔类型（boolean type）”的概念，关键字中也没有“true”和“false”，有的只是关系运算和逻辑运算，而这两种运算的结果都是整型值，要么是“0”（代表“假”），要么是“1”（代表“真”）。例如：

```
int a = 1;
int b = 2;
```

则有：

（a < b）返回“1”；

（b < a）返回“0”；

（(a < b) && (a > 0)）返回“1”；

（(b < a) || (a < 0)）返回“0”。

对于参与逻辑运算的表达式，只要其值不为“0”就认为它是“真”，就好像它的值是“1”那样。例如：

```
int a = -4;
int b = 0;
```

（a && b）返回“0”；

（a || b）返回“1”。

一元运算符“!”也是如此：

```
int a = 5;
int b = 0;
```

（! a）返回“0”；

（! b）返回“1”；

要是涉及到赋值语句，则编译器会自动安排类型转换，我们不用干预：

```
int a = 1;
```

```
int b = 0;
int i = (a > b);                                    /* OK */
char c = (b > a);                                   /* OK */
double d = ((a < b)||(b > 0));                      /* OK */
```

C++就不一样，它有布尔类型，用关键字“bool”标识：

```
bool b = 1;                    /* 定义布尔变量 b */
```

同时“true”和“false”也是关键字，代表内置的两个布尔常量：“真”和“假”。C++编译器都用数值“1”表示“true”，用数值“0”表示“false”。

下面是true和false的用法：

```
bool b = true;                          /* 初始化 bool 变量 b 为“真”值 */
bool nb = ! b;                          /* 对 b 求反然后赋给 nb */
```

现在，C99也跟上潮流，增加了布尔类型，用关键字“_Bool”标识：

```
_Bool b;                                /* 声明_Bool 变量 */
```

_Bool变量占用一个字节，它只有两个取值“1”和“0”，分别代表“真”和“假”（“true”和“false”仍然不是C99的关键字）。我们不妨把_Bool变量看成是一种特殊的整型变量。

由于有_Bool变量专门去表示“真”和“假”，关系运算和逻辑运算的表达也就更加自然：

```
int a = 0, b = 1;
_Bool b = (a < b);
_Bool b2 = ((! b)||(b > 0));
```

不过，以往很多C代码都通过使用预处理指令定义了布尔类型，例如：

```
#define bool int
#define true 1
#define false 0
```

有了上面的宏，我们在 C90 里可以这样写：

```
bool b = true;
```

经过预处理，这条式子其实是：

```
int b = 1;
```

由于 C++处理关系运算和逻辑运算的规则和 C 是一样的，所以上面的代码完全可以通过 C++编译器的编译。

现在，C 标准委员会建议：如果要使用“bool”、“true”和“false”，最好使用 C99 新增加的标准头文件 stdbool.h，以增强代码的移植性。

让我们看看 gcc 的 stdbool.h：

```
H01    #ifndef  _STDBOOL_H
H02    #define  _STDBOOL_H
H03
H04    #ifndef  __cplusplus
H05    #define  bool  _Bool
H06    #define  true  1
H07    #define  false  0
H08    #else
H09    #define  _Bool  bool
H10    #endif
H11
H12    #define  __bool_true_false_are_defined  1
H13
H14    #endif
```

H01、H02 和 H14 是防止重复包含头文件的常见手法。

H04 判断源文件将要被什么编译器编译，如果是 C 编译器，则由于 C 的预处理程序不会定义“__cplusplus”宏，于是预处理程序根据 H05、H06 和 H07 进行宏替换；

如果是 C++编译器，则由于 C++预处理程序一定会定义“__cplusplus”宏，这时预处理程序就按照 H09 进行宏替换。

H12 设置信号标志，表示“bool”、“true”和“false”已经被定义。

我们讨论一下 stdbool.h 的用法和作用：

如果我们只是编写“纯粹的”C 代码（pure C code）[1]，那么根本不需要包含 stdbool.h。并且，还可以把“bool”、“true”和“false”作为普通标识符使用（因为它们不是 C 语言的关键字）：

```
int bool = 1;                    // 尽管有些丑陋，但却是完全正确的 C 代码
```

不过，一旦真的把“bool”、“true”和“false”用作标识符，就决不能包含 stdbool.h（看看 H05、H06 和 H07 就知道为什么）。

如果我们的 C 代码像 C++那样去使用“bool”、“true”和“false”，不过是用以前自己写的头文件实现的，那么，最好把自己写的头文件去掉，然后包含 stdbool.h。

C99 只要求 stdbool.h 能够帮助 C 编译器认识“bool”、“true”和“false”，所以，上面 gcc 的 stdbool.h 即使省略 H09 也是符合 C 标准的。不过，为了让 C++编译器（通过宏替换）也能认识“_Bool”，H09 作为扩展功能一般都会被加上去。这样，只要包含 stdbool.h，使用“_Bool”的 C 代码同样能被 C++编译器编译。[2]

[FN3001]：“纯粹的”C 代码是指那些根本不要求通过 C++编译器编译的 C 代码。

[FN3002]：有兴趣的读者可以进一步参阅[Meyers, 2002]。

31 _Complex 与 _Imaginary

虽然C90没有直接提供对复数及其运算的支持，但各种C编译器都有自己的扩展，通过头文件complex.h[1]里面的宏以及编译器内置的相应功能，我们早就能够在C代码中使用复数。

C99正式增加了复数和虚数类型，分别用关键字“_Complex”和“_Imaginary”标识[2]。就像int有short、int、long一样，_Complex和_Imaginary其实各自有3种更具体的类型：

```
float  _Complex          float  _Imaginary
double  _Complex         double  _Imaginary
long  double  _Complex   long  double  _Imaginary
```

这很好理解，复数是由实部、虚部和虚数单位组合而成，而虚数单位是常数，所以实部和虚部用什么类型的浮点数表示就构成什么类型的复数，例如：

```
float _Complex z = 1.0F + I * 2.0F;        /* I是虚数单位 */
```

上面定义一个单精度浮点复数。

又或者：

```
_Complex z = 5.89 - I * 6.21;
_Imaginary z = I * 7.8;
```

不明确指出浮点类型的_Complex、_Imaginary被默认为是double _Complex、double _Imaginary。

C99增加标准头文件complex.h，里面有关于复数的宏定义和运算函数。只要包含这个标准头文件，就可以在C代码中进行规范的复数运算处理：

```
C01    /* Example code 31-01, file name: 3101.c */
C02
C03    #include<stdio.h>
C04    #include<complex.h>
C05    int main(void)
C06    {
C07        _Complex a = 2.0 + I * 4.0;
C08        _Complex b = 3.0 + I * (-2.0);
C09        _Complex c = a * b;
C10        printf("c = %+f%+fi\n", creal(c), cimag(c));
C11        return 0;
C12    }
```

```
$gcc  -std=c99  3101.c
$./a.out
c = +14.000000+8.000000i
$
```

以往的大部分C编译器在扩展功能时已经用“complex”表示复数类型，如果我们的C代码使用了“complex”，那么现在规范的做法应该是包含标准头文件complex.h。下面是gcc中complex.h其中的一小段：

```
...
#define  complex  _Complex
#define  _Complex_I  (__extension__1.0iF)
#undef  I
#define  I  _Complex_I
...
```

可以看出，头文件通过宏把“complex”替换成“_Complex”，并且定义了虚数单位“I”。[3]

由于C++通过模板类std::complex实现复数，所以，C99和C++在这方面基本

上无法“沟通”。使用复数的 C99 代码不能被 C++编译器编译，使用复数的 C++代码同样无法被 C 编译器编译。下面给出例子 31-01 的 C++版本，大家可以对照一下：

```
C01     /* Example code 31-02, file name: 3102.cpp */
C02
C03     #include<iostream>
C04     #include<complex>
C05     using namespace std;
C06
C07     int main(void)
C08     {
C09     complex<double>  a(2.0, 4.0);
C10     complex<double>  b(3.0, -2.0);
C11     complex<double>  c = a * b;
C12     cout.setf(ios::showpos);
C13     cout.setf(ios::fixed);
C14     cout << "c = " << c.real();
C15     cout << c.imag() << "i" << endl;
C16        return 0;
C17     }
```

```
$g++  3102.cpp
$./a.out
c = +14.000000+8.000000i
```

[FN3101]: complex.h 不是 C90 的标准头文件，只是各种编译器为了实现自己的扩展而添加的，具体的名字可能在不同的平台上有所不同。

[FN3102]：C99 指出，对虚数类型“_Imaginary”的支持是可选的。就是说，C99 编译器可以仅仅支持复数类型“_Complex”而不支持“_Imaginary”，因为如果编译器支持_Complex，实际上也就间接地支持了_Imaginary。别忘了，虚数是复数的子集。

[FN3103]：各种编译器对虚数单位的表达不一定都相同，这里仅仅是 gcc 的写法。

32 内联函数

在内联函数（inline function）出现之前，如果我们想提高程序的运行效率，可以适当使用宏。例如有一个函数，如果它会被频繁地调用，为了节省函数调用过程中的系统开销，我们可以把函数写成宏：

```
#define  abs(x)  ((x) >= 0) ? (x): -(x)
```

这样，（通过预处理程序展开宏）我们每一次调用 abs(x)，譬如：

```
int a = abs(-5);
```

就相当于直接写：

```
int a = ((-5) >= 0) ? (-5): -(-5);
```

效率明显比编写一个函数然后调用它要好。

但宏也不是万能的，它有很多不尽人意的方面，例如：

```
int a = abs(5) + 1;                    /* 结果 a 应该等于 6 */
```

然而，预处理程序把宏展开：

```
int a = ((5) >= 0) ? (5): -(5) + 1;
```

“+”运算符的优先级比“？”要高，所以上面的式子其实是：

```
int a = ((5) >= 0) ? (5): (-(-5) + 1);        /* a=5!  */
```

为了避免这样的漏洞，我们要小心翼翼地提防，尽可能采取适当的措施：

```
#define  abs(x)  (((x) >= 0) ? (-x): -(-x))
```

虽然这里“加括号”可以解决问题，但由于宏具有不少“副作用”，某些场合就颇为棘手。[1]

不妨想一想：宏为什么会有这些缺陷？

——因为宏仅仅是一种记号替换，不像函数，它没有办法知道真正的参数是什么，从而无法对参数进行检查，也不能像函数那样把某些运算局限于内部。

如果有一种函数，它本身的确是函数，但必要时编译器可以把它的代码像宏那样“嵌

入”到调用它的地方，从而既节省系统开支又没有宏的缺点，那就好了。

这就是“内联函数”的概念，最先是由C++提出和实现的，到了今天，C99终于也拥有了这项语言特性。

我们先从C++的角度看看内联函数究竟是怎么回事[2]：

```
C01    /* Example code 32-01, file name: 3201.cpp */
C02
C03    inline int fun(int a);
C04    int main(void)
C05    {
C06        int a = 0;
C07        int b = fun(a);
C08        return 0;
C09    }
C10
C11    int fun(int a)
C12    {
C13        return ++a;
C14    }
```

C03声明一个内联函数fun，这个函数的功能很简单，它仅仅返回参数的自增值，我们看看汇编代码[3]：

```
$g++ -S 3201.cpp -finline-functions
$less 3201.s
```

```
A01        .globl main
A02        .type  main,@function
A03        main:
```

```
A04     ...
A05     movl   $0, -4(%ebp)
A06     movl   -4(%ebp), %eax
A07     movl   %eax, -12(%ebp)
A08     leal   -12(%ebp), %eax
A09     incl   (%eax)
A10     movl   -12(%ebp), %eax
A11     movl   %eax, -16(%ebp)
A12     movl   -16(%ebp), %eax
A13     movl   %eax, -8(%ebp)
A14     ...
```

大家有没有发现类似：

```
call  fun
```

的函数调用语句？

没有。因为实现函数功能的代码已经“嵌入”到 main 函数里面，而且不像普通函数，编译器根本就没有产生一个全局符号，也就是说，其他文件的代码是不能够调用 fun 的。

由于内联函数要求编译器在调用点尽可能地嵌入函数的代码，所以，仅仅声明一个内联函数却没有在同一文件中给出函数定义的做法是没有意义的。C++明确规定，一旦使用 inline 声明一个函数，这个函数就成为内联函数，所有调用该函数的文件都应该（在同一文件中）给出函数的定义，并且所有这些定义都应该是相同的。所以，内联函数的定义一般都直接放在头文件里面。

C++的 inline 修饰符不改变函数的链接特性，所以内联函数完全可以有外部链接特性。C++进一步规定，所有外部链接的内联函数都必须具有相同的地址，它们内部定义的静态变量都指向同一个实体。

inline 关键字仅仅是建议编译器尽可能嵌入代码，编译器根据实际情况完全可以决定是否采纳建议。有时候编译器仍然会像普通函数那样为内联函数生成一段函数代码和符号，譬如编译器认为你的函数太复杂而无法嵌入，这时它就会把内联函数当作普通

函数那样进行调用（用 call 指令）；又譬如程序员定义了函数指针，然后把内联函数的地址赋给指针，这种情况也要求编译器为内联函数生成符号从而使链接程序对其分配地址。但 C++又要求每一个使用内联函数的文件都要有函数的定义，这就导致内联函数可能会在多个文件内被定义，一般情况下链接程序不允许出现这种情况，因为这样会导致名字冲突。

C++编译器怎样解决这个问题呢？如何让可能存在的多份内联函数副本正常链接、具有相同的地址？g++的方案很简单，就是修改链接程序，对内联函数执行特殊政策，链接程序不会去“计较”内联函数存在多份副本的事实而仅仅对其中的一份副本分配地址然后完成链接。

```
$g++  -S  3201.cpp
$less  3201.s
```

```
A01       .globl main
A02       .type  main,@function
A03       main:
A04       ...
A05       call   _Z3funi
A06       ...
A07
A08       .section  .gnu.linkonce.t._Z3funi,"ax",@progbits
A09       .weak  _Z3funi
A10       .type  _Z3funi,@function
A11       _Z3funi:
A12       ...
```

C++编译器生成的符号和 C 编译器不同，我们暂时不详细讨论这个问题，现在只要大家知道上面的“_Z3funi”就是 g++为 32-01 的函数 fun 生成的符号。注意 A08 和 A09，除了这两句，其他代码和平时定义普通函数是一样的。如果 fun 不是内联函数，则这两句不会出现，取而代之的是：

```
.globl _Z3funi
```

现在，A08、A09 的意思是指出符号_Z3funi 可能会在多个目标代码文件中出现，随后链接程序就会知道只对其中一份的副本进行地址计算和链接。

基于这种实现，链接程序不会去检测（在多个文件中定义的）内联函数是否完全一样，即使函数定义存在不一致，代码也会链接成功。事实上，C++标准明确指出不需要实现平台进行这种检测。看看下面的例子：

```
C01    /* Example code 32-02, file name: 3202.cpp */
C02
C03    #include<iostream>
C04    using namespace std;
C05    inline void f()
C06    {
C07    cout << "f() in 3202.cpp" << endl;
C06    }
C07    void g();
C08    int main(void)
C09    {
C10        void (*pf)();
C11        pf = f;
C12        pf();
C13        f();
C14        g();
C15    }
```

```
C01      /* Example code 32-02a, file name: 3202a.cpp */
```

```
C02
C03         #include<iostream>
C04         using namespace std;
C05         inline void f()
C06         {
C07         cout << "f() in 3202a.cpp" << endl;
C08         }
C09         void g()
C10         {
C11         f();
C12         }
```

```
$g++  3202.cpp  3202a.cpp
$./a.out
f() in 3302.cpp
f() in 3302.cpp
f() in 3302.cpp
$
$g++  3202a.cpp  3202.cpp
$./a.out
f() in 3302a.cpp
f() in 3302a.cpp
f() in 3302a.cpp
$
```

以上的例子表明，即使内联函数的定义不一致，链接程序也不报错（因为它根本就没进行相关检测），并且在某种情况下[4]，不同的链接顺序会得出不同的结果。

第一次我们的链接顺序是 3202.o（先）、3202a.o（后）。链接程序找到的第一个满足要求的_Z1fv（即函数 f）位于 3202.o，因此所有对_Z1fv 的调用都是对 **3202.o**

的_Z1fv 的调用。

而第二次我们的链接顺序是 3202a.o（先）、3202.o（后）。链接程序找到的第一个满足要求的_Z1fv（即函数 f）位于 3202a.o，因此所有对_Z1fv 的调用都是对**3202a.o** 的_Z1fv 的调用。

自然，两次不同链接顺序产生的可执行文件是不同的，运行结果也就不一样了。

上面仅仅是为了说明道理而举的例子，一般情况下由于内联函数都统一放在头文件，只要我们保证包含同一个头文件就可以确保分布在各个源代码文件中的内联函数具有一致的定义。

有了解决内联函数链接问题的方案，接着再处理内联函数里面的静态内部变量就很容易。前面提到，C++标准要求同一个内联函数的静态内部变量都指向同一个实体。C++编译器同样会指示链接程序对内联函数的静态内部变量进行类似的处理，虽然存在静态变量的多份副本，但链接程序仅仅链接其中的一份。稍有不同的是，C++编译器必须根据某种策略来为这些静态变量生成符号，这个策略既要保证生成的符号不会与其他的变量产生冲突，又要保证为分布在不同文件的静态变量所生成的符号都是相同的。只有这样，下一步链接程序才能正确地进行链接。

g++的办法同样很简单，只要把内联函数的符号信息“添加”进去就可以了：

```
C01    /* Example code 32-03, file name: 3203.cpp */
C02
C03    inline int fun()
C04    {
C05        static int var = 0;
C06        return (++var);
C07    }
C08
C09    int main(){     int count = fun();      }
```

```
$g++  -S  3203.cpp
$less  3203.s
```

```
A01         ...
A02         .weak  _ZZ3funvE3var
A03         .section
            .gnu.linkonce.d._ZZ3funvE3var,"aw",@progbits
A04         .align 4
A05         .type  _ZZ3funvE3var,@object
A06         .size  _ZZ3funvE3var,4
A07     _ZZ3funvE3var:
A08         .long  0
A09
A10         .section
            .gnu.linkonce.t._Z3funv,"ax",@progbits
A11         .align 2
A12         .weak  _Z3funv
A13         .type  _Z3funv,@function
A14     _Z3funv:
A15         ...
```

上面的“_Z3funv”是g++为内联函数fun生成的符号，而对应fun的内部静态变量var的符号是“_ZZ3funvE3var”。很明显，后者包含了前者的信息在内。采取这样的命名策略之后，C++编译器就可以保证为内联函数的内部静态变量生成的符号既惟一又一致。当然，前提是内联函数的定义本身是既惟一又一致的[5]，然而这是程序员自己的责任了。

C99对内联函数的规定与C++相比，既有相同的地方又有不小的差别。

相同的是编译器有权根据实际情况决定是把内联函数的代码嵌入还是按照普通函数那样用call指令调用；不同的是，C++没有明确指出编译器在“嵌入代码”成功之后

是否仍然为内联函数生成符号，一般地，程序员通过编译器提供的相关编译选项加以控制，而C99在这个问题上对具有外部链接性质的内联函数有明确规定，下面我们来讨论这个问题。

在C99中，一旦用inline声明一个函数，这个函数就成为内联函数。内联函数必须在同一文件中定义。例如：

```
...
inline int fun(void);                    //声明fun是内联函数
...
int fun(void){ ... }                     //必须在同一文件给出fun的定义
...
```

又或者：

```
...
inline int fun(void){ ... }              //在声明的同时给出定义
...
```

对于所有内部链接的函数来说，成为内联函数是绝对没问题的，例如：

```
...
inline static int fun(void) { ... }
...
int main(void){   fun();  }
```

对于这种情况，C99编译器首先会尝试嵌入代码。如果嵌入成功，则编译器自行决定是否为内联函数生成符号（程序员可以通过编译选项加以控制）。万一嵌入失败，则C99编译器必须像对待普通的静态函数一样，生成局部可见的符号及函数代码，（回忆一下以前讲的静态函数对应的汇编代码）大致如下：

```
    .type  fun, @function          //注意：没有“.globl fun”这一句
fun:
    ...
```

然后通过 call 指令调用函数。由于 fun 是局部可见的符号，所以不会和其他文件的同名函数发生任何名字冲突。这些都是和 C++一样的。

但对于外部链接的函数，C99 和 C++就不同了，例如：

```
...
inline int g(void){  ...  }      //把外部链接的函数声明为内联函数
...
int main(void){  g();  }
```

C99 规定：除非在（同一文件内）用 extern 显式声明函数，否则编译器不会为内联函数生成相应的符号，无论代码嵌入成功与否。

像上面举的例子，C99 编译器要么成功地把函数 g 的代码嵌入，要么用 call 指令调用 g，但无论如何，编译器就是不会为内联函数 g 生成符号。我们可以肯定在汇编代码中不会出现下面的语句：

```
.globl g
    .type g, @function
g:
    ...
```

而且，C99 对外部链接的内联函数有一个限制：不能定义静态内部变量（除非用 const 修饰）以及不能访问具有内部链接性质的变量和函数。例如：

```
...
static int a = 0;
static int f(void){ ... }
...
inline int g(void)
{
    static int s = 1;                //错误，定义了静态内部变量
    const static int c = 1;          //正确，加上 const 就没问题
```

```
    ++a;                                    //错误，访问了内部链接性质的变量
    f();                                    //错误，访问了内部链接性质的函数
}
```

C99 为什么要这样规定呢？

笔者猜测是为了尽量避免改动以往的链接程序。前面已经分析过，如果采取 C++的做法，则为了避免名字冲突我们就必须额外修改已有的链接程序，标准 C 委员会目前可能不认为这是一个好主意。于是，对于外部链接的内联函数，C99 解决名字冲突的方案是程序员如果需要为内联函数生成符号则必须在 C 源代码文件中给出明确的指示：

```
...
inline int f(void){ ... }           //外部链接的函数声明为内联函数
...
extern int f();                     //指示编译器为内联函数生成符号
...
int main(void)
{
    f();                            //可以嵌入代码，也可以用 call 调用
}
```

上面的代码编译成汇编后会出现下列我们熟悉的语句：

```
.globl f
    .type f, @function
f:
    ...
```

因为我们明确用 extern 声明函数 f，所以 C99 编译器会为 f 生成相应的符号及代码。在这种情况下，程序员同样要自己来保证在其他文件中没有同名的全局函数。

现在，我们同样可以将内联函数的定义放在头文件，然后在某一个 C 文件中明确指示生成符号：

decla.h

```
#ifndef DECLA_H
#define DECLA_H
extern void fun(void);
#endif
```

defin.h

```
#ifndef DEFIN_H
#define DEFIN_H
inline void fun(void)
{
    ...
}
#endif
```

Main.c

```
#include"defin.h"
int main()
{
    fun();
}
```

fun.c

```
#include"defin.h"
#include"decla.h"
```

上面的main调用了内联函数fun，无论嵌入成功与否，编译器都不会为fun生成符号，所以我们可以完全放心地在fun.c中指示编译器生成符号。[6]

[FN3201]：有兴趣的读者请参阅[Koenig, 1989]。

[FN3202]：先讲述C++的内联函数然后再讨论C99的内联函数可能会比较好。

[FN3203]：请注意，下面的这一段汇编代码是用g++ 3.0.4版生成的，因为g++ 3.2版由于某些原因暂时取消了把内联函数嵌入到代码中的做法。为了让大家了解什么是嵌入代码，这里以旧版g++产生的汇编作为例子，但仅限于这一段，本节的其他汇编代码仍然是用3.2版的gcc或g++编译。

[FN3204]：这里的“某种情况”是指内联函数没有被嵌入代码（因为函数太复杂或者程序员指示编译器这样做），而是像普通函数那样通过call指令调用。

[FN3205]：这里说内联函数的定义惟一，并不是指内联函数只能有一处定义，而是指（在本文件或其他文件内）不能存在和内联函数的名字和参数都相同的普通函数。

[FN3206]：有兴趣的读者请参阅[Meyers, 2002a]。

33 变长数组（上）

在C90和C++里定义数组，方括号内必须是编译期可求出的常数值，例如：

```
int array[10];                  /* OK both in C90 and C++ */
int array[10/5+6];              /* ok both in c90 and c++ */
```

或：

```
const int SIZE = 50;
int array[SIZE];                /* OK only in C++ */
```

现在，C99迈出一大步，允许方括号里面是整型变量或整型表达式，这样的数组称为“变长数组”（Variable Length Array）。注意，变长数组是指用（整型）变量或表达式声明（或定义）的数组，而不是说数组的长度会随时变化，变长数组在其生存期内的长度同样是固定的。

下面是简单的例子：

```
int f(void);
int main(void)
{
    int size = 4;
    int a[size];                // OK only in C99
    int b[size*size];           // OK only in C99
    int c[f()];                 // OK only in C99
    ...
}
```

代码中的size是整型变量，（size*size）是整型表达式，f()是返回int的函数，C99允许我们用它们来作为数组的长度去声明或定义数组。

为什么C99要支持变长数组这项特性呢？

主要是为了让C程序员更方便地使用多维数组。以前，如果在程序运行的时候才能

确定一个多维数组的具体大小，那就只能用一维数组去“模拟”。例如我们要使用一个三维数组 int p[l][m][n]，而 l、m、n 的值无法在编译的时候确定，就可以这样写：

```
...
int i, j, k, l, m, n, *p;
int fun(int, int, int);
...                                          //确定 l、m、n 的值
p = malloc(l * m * n * sizeof(int));         //动态分配内存
for(i = 0; i < l; ++i)
    for(j = 0; j < m; ++j)
        for(k = 0; k < n; ++n)
            p[i*m*n+j*n+k] = fun(i, j, k);   //必须正确计算偏移
...
```

可以看到，如果用一维数组模拟三维数组，那么正确计算元素的偏移值就成为程序员的额外负担，像上面的最后一条语句，写法繁琐、不直观，频繁使用时还很容易出错。

有了变长数组后，情况大大改善：

```
...
int l, m, n;
int fun(int, int, int);
...                                          //确定 l、m、n 的值
int array[l][m][n];                          //使用 VLA
for(int i = 0; i < l; ++i)
    for(int j = 0; j < m; ++j)
        for(int k = 0; k < n; ++n)
            array[i][j][k] = fun(i, j, k);   //不用我们去计算偏移
...
```

跟使用编译期可知长度的数组一样，正确计算地址值是编译器的责任，我们只须简

单地使用 array[i][j][k]就能访问任何一个元素。

在使用变长数组的过程中要注意一些问题：

#1 关于 sizeof

以往，在程序编译时我们就已经可以知道 sizeof 运算符的返回结果。例如：

```
int array[10];
printf("Array has %d elements.\n", sizeof(array)/sizeof(int));
```

在编译的时候，printf 语句已经变成：

```
printf("Array has %d elements.\n", 40/10);
```

如果使用变长数组，可以猜到，sizeof 运算的结果要在程序实际运行时才能求出。

#2 不能声明（或定义）文件范围的外部变长数组。由于外部数组存放在数据段，它们占用的空间必须在编译期确定，所以变长数组不能在函数外面声明或定义：

```
extern int n;
int a[n];                    // Error: VLA 不能在函数外面定义
int main(void)
{
    int m;
    ...
    static int b[m];         // Error: VLA 不能是静态的
    extern int c[m];         //Error: VLA 不能有外部链接性质
    ...
}
```

#3 变长数组不能是 struct 或 union 的成员，因为 struct 和 union 的大小必须在编译期确定：

```
struct s
{
```

```
    int n;
    int array[n];              // Error
};
```

#4 必须保证变长数组的方括号中的表达式返回整型量而且大于“0”，否则，后果是不确定的：

```
int n = -5;
int array[n];                  // 危险的代码，后果难料！
```

#5 方括号里的表达式的求值顺序是不确定的：

```
int f(), g(), h();
int array[f()][g()];
int array2[h()];
```

上面的代码中，f()可能先于 g()调用，也可能相反；惟一可以肯定的是，h()在f()和g()都调用完毕之后再调用。

34 变长数组（下）

这一节我们继续讨论变长数组的特性。

首先，正如前面已经提到的，变长数组虽然要到运行期才能确定具体占用存储空间的多少，但这并不意味变长数组的长度会随时改变：

```
C01    /* Example code 34-01, file name: 3401.c */
C02
C03    #include<stdio.h>
C04    void fun(int n);
C05    int main(void)
C06    {
C07        fun(10);
C08        return 0;
C09    }
C10
C11    void fun(int n)
C12    {
C13        int vla[n];
C14        n += 10;                          // n is modified!
C15        printf("vla takes %d Bytes.\n", sizeof(vla));
C16    }
```

```
$gcc -std=c99 3501.c
$./a.out
vla takes 40 Bytes.
$
```

大家可以看到，尽管 C14 对 n 进行了修改，但 C13 声明的 vla 却不会因此而改变大小，因为变长数组一旦被声明，其大小就会保持不变直到生存期结束。以下代码更能说明问题：

```
C01    /* Example code 34-02, file name: 3402.c */
C02
C03    #include<stdio.h>
C04    void fun(int n);
C05    int main(void)
C06    {
C07        fun(10);
C08        return 0;
C09    }
C10
C11    void fun(int n)
C12    {
C13        typedef int VLA[n];
C14        VLA a;
C15        n += 10;                    // n is modified!
C16    VLA b;
C17        int c[n];
C18        printf("a takes %d Bytes.\n", sizeof(a));
C19    printf("b takes %d Bytes.\n", sizeof(b));
C20        printf("c takes %d Bytes.\n", sizeof(c));
C21    }
```

```
$gcc -std=c99 3402.c
$./a.out
a takes 40 Bytes.
b takes 40 Bytes.
c takes 80 Bytes.
$
```

通过 typedef 声明的变长数组 a 的大小是 40 字节，这是毫无疑问的。关键是数组 b，它的大小仍然是 40 字节，因为 b 是由 VLA 声明的，而定义 VLA 的 typedef 语句是 C13，所以，一切通过 VLA 声明的变长数组的大小都以（程序）执行到 C13 时的 n 值为依据进行计算确定，在上例中，C07 把整数 10 传给函数 fun，从而 C13 所定义的 VLA 的大小是 40 字节。

而 C17 的变长数组 c 由于不是通过 VLA 声明的，所以 c 的具体大小由序执行到 C17 时的 n 值确定，在例子中，当时 n 等于 20，所以 c 的大小是 80 字节。

（编译器生成的）代码之所以能够如此神奇地记住变长数组在最初声明时的具体长度，主要是通过增加（隐藏的）临时变量实现的。最简单、最原始的办法可以是每遇到一个变长数组就插一个临时变量以存储数组的真实大小，例如：

```
...
int vla[n];
n += 10;
printf("a takes %d Bytes.\n", sizeof(vla));
...
```

被处理成：

```
...
int temp = n;
int a[temp];
n += 10;
printf("a takes %d Bytes.\n", (temp * 4));
...
```

通过临时变量，变长数组的长度就可以在数组的生存期内被保存起来供使用者随时访问确认。

和普通数组一样，多维变长数组和指向变长数组的指针之间同样有对应关系：

```
C01    /* Example code 34-03, file name: 3403.c */
C02
C03    int main(void)
C04    {
C05        int n = 10;
C06        int vla[n][n];
C07        int (*pvla)[n] = vla;
C08        for(int i = 0; i < n; ++i)
C09            for(int j = 0; j < n; ++j)
C10                pvla[i][j] = i * j;
C11        return 0;
C12    }
```

C06 定义了二维变长数组 vla，C07 定义了指向变长数组的指针 pvla，并且把 vla 的地址作为初始值赋给 pvla，然后 C10 就可以通过 pvla 访问 vla。

不过，在使用指向变长数组的指针时，同样需要注意数组类型是否相同，例如：

```
int n, m;
int vla[n][6][m];
int (*pvla)[4][m];
pvla = vla;                    // Error: 4 != 6
```

又例如：

```
int n, m;
int vla[n][n][6][m];
int (*pvla)[n][n][n+1];
pvla = vla;
```

代码可以通过编译，但只有当 n 等于 6 而且 m 等于 n+1 时，代码才能正常工作，否则，后果难料。

最后，在函数原型中的变长数组可以用“*”代替表达式，例如：

```
int fun(int n, int m, int vla[n][m]);
```

和

```
int fun(int n, int m, int vla[*][*]);
int fun(int n, int m, int vla[ ][*]);
int fun(int n, int m, int vla[ ][m]);
```

全部都是等价的，编译器把它们看作是相同的函数原型。

不过，在函数定义中就一定要写出表达式而不能用“*”代替了

最后要注意的是，以上所说关于 VLA 的一切，C++都未提供任何支持。[1]

[FN3401]：有兴趣的读者可以进一步参阅[Meyers, 2001b]。

35 可伸缩数组成员

C99 对于 struct 的使用作出了某些改进，使得我们能够更加方便地定义、初始化和使用 struct 类型的变量。

以前，我们可能会碰到这样的问题：如何定义一个包含可变长度数组的 struct 呢？例如，我们想用一个 struct 存放某个平面多边形的各个顶点的坐标，然后计算它的面积或各条边长等。显然，完全可以用一个浮点数组存储这些坐标值，关键是用户输入的多边形可以是三角形、四边形或 N 边形，总之，边的数目是运行时才能确定的。由于 struct 占用的存储空间必须在编译期确定，因此我们肯定不能这样写：

```
struct polygon
{
    int edgeCount;
    double coordinate[edgeCount * 2];     // Error!
};
```

因为 struct 变量不能包含变长数组，变通的办法通常是：

```
struct polygon
{
    int edgeCount;
    double coordinate[2];                 //先用一个点占着地盘
};
```

定义的时候先安排一个顶点的坐标“进驻”polygon，实际使用的时候才通过动态分配内存放入所有的顶点坐标：

```
...
struct polygon *p;
int temp, i = 0;
...                          //让用户输入多边形的边数，检验合法性后放入 temp
p = malloc(sizeof(struct polygon) + 16 * (temp - 1));[1]
...
p->edgeCount = temp;                    //放入多边形的边数
while(temp--)
{
    p->coordinate[i++] = ...        //放入第 i 个顶点的 X 坐标
    p->coordinate[i++] = ...        //放入第 i 个顶点的 Y 坐标
}
//现在 p 指向的内存已经被成功初始化，可以在后面的运算中使用
...
```

之所以能够耍出这样的小花招，靠的是 C 语言的“豪放”——不提供数组越界的运行期检查，编译器对数组元素的访问仅仅是通过将数组起始地址和偏移量相加得到其地址的，至于这个地址是否已经超出范围，编译器可不管，看：

```
int array[10];
int i = 78;
array[22] = 5;                          //完全可以通过编译
array[i] = 8;                           //同上
```

不过，由于硬是要“塞进”至少一个数组元素，上面的 polygon 定义始终不那么自然，于是，C99 通过允许 struct 包含“Flexible Array Memeber”（可伸缩的数组成员），让你索性“把花招进行到底”：

例如：

```
struct polygon
{
    int edgeCount;
    double coordinate[ ];              // OK in C99!
};
```

上面的定义告诉 C99 编译器，polygon 有一个可伸缩数组成员：coordinate，它不占用任何空间。例如，一个很明显可以观察到的事实是：

sizeof(polygon)等于 sizeof(edgeCount)

另外，C99 编译器在所有涉及到(静态)存储分配的时候(例如定义外部的 polygon 变量，或者在函数里声明内部的 polygon 变量，又或者把某个 polygon 变量作为参数传递给函数等)，都“认为” polygon 只包含一个 edgeCount 成员变量。这是很自然的结果，因为在 polygon 定义中根本没有指明数组 coordinate 的大小。仅仅当引用 coordinate 的时候，编译器负责算出具体的偏移地址，剩下的事情完全由程序员负责(例如这个地址是否有效等)。

由于在静态存储分配时编译器不会为 coordinate 分配任何空间，所以只有在进行了动态存储分配后，使用 coordinate 才有意义。下面是 C99 版本“多边形程序”的概要：

```
...
struct polygon
{
    int edgeCount;
    double coordinate[ ];
};
...
struct polygon *p;
int temp, i = 0;
...                              //让用户输入多边形的边数，检验合法性后放入 temp
p = malloc(sizeof(struct polygon) + 16 * temp);
```

```
...
p->edgeCount = temp;                    //放入多边形的边数
while(temp--)
{
    p->coordinate[i++] = ...            //放入第 i 个顶点的 X 坐标
    p->coordinate[i++] = ...            //放入第 i 个顶点的 Y 坐标
}
//现在 p 指向的内存已经被成功初始化，可以在后面的运算中使用
...
```

在 struct 中使用可伸缩数组成员时要注意一些问题：

#1 （数组）元素地址的有效性完全由程序员负责，编译器是不管三七二十一的，类似下面的代码虽然错得离谱却能顺利编译：

```
void fun()
{
    struct { int n; double array[ ]; } s;
    ...
    s.array[4] = 3.45;                  //根本没有通过指针动态分配内存
    ...
}
```

s 只有一个成员，它就是 int 类型的 n，double 数组 array 是根本不存在的，我们也没有进行任何动态存储分配，但编译器却无法检查出上述代码的错误。

#2 包含可伸缩数组成员的 struct 至少要另外包含一个编译期可确定大小的对象，并且可伸缩数组成员在 struct 中必须是最后一个成员。

这条规则很容易理解，由于在所有场合（除非执行动态存储分配），可伸缩数组成员实际上都不占用空间，所以如果 struct 不至少另外包含一个大小确定的对象，则相当

于定义一个长度为零的 struct，这当然是编译器不允许的。

可伸缩数组成员必须被安排放到 struct 的最后也是很显然的，试想一下，可不可能有这样的定义：

```
struct demo
{
    int n;
    char array[ ];                          //Error
    double d;
};
```

——编译器该认为成员 d 的偏移是多少？！

同理，可伸缩数组成员不能多于一个：

```
struct demo
{
    int n;
    char array[ ];
    double array2[ ];                       //Error
};
```

——编译器又傻眼了……

#3 一定要记住，不管任何时候，压栈和赋值操作都不会把可伸缩数组成员包含在内。

例如：

```
struct demo
{
    int n;
    char array[ ];
};
...
struct demo *p1, *p2;
```

```
...                                          //对 p1、p2 执行内存动态分配
*p1 = *p2;                                   //赋值操作！究竟赋多少个字节？
```

好，上面最后一句代码执行的结果是什么呢？假设 p2->array 由于刚刚成功分配到内存而占用 100 个字节，则“*p1 = *p2;”是否意味着这 100 个字节的内容也被拷贝到 p1->array 呢？

答案是否定的。

*p1 = *p2;

的操作结果仅仅是：

```
p1->n = p2->n;
```

同理，函数参数压栈也是如此：

```
struct demo
{
    int n;
    char array[ ];
};
...
void fun(struct demo s);
...
struct demo *p;
p = malloc(sizeof(struct demo) + 100);     //假设申请成功
fun(*p);                                   //压栈的仅仅是 p->n
```

#4 由于在动态分配内存之前可伸缩数组成员不占用任何存储空间，所以我们无法对可伸缩数组成员进行初始化：

例如：

```
struct demo
{
    int n;
    double array[ ];
};
...
struct demo s1 = {3, {2.5, 7.8}};                    //Error!
struct demo s2 = {5};                                //OK!
```

36 Designated Initializer 和 Compound Literal

C99 对于复合类型（struct、union 和 array）的初始化有新的、更为灵活的语法。以往，我们如果要求只对数组的某个元素或结构体、共用体的某个成员进行初始化，通常被迫采用比较笨拙的写法，很容易出错，有时甚至是不可能的，例如：

```
int array[10] = {0, 0, 0, 0, 2};    //仅仅把 array[4]初始化为 2
```

如果再刁钻一些，我们要把 array[0]、array[2]、array[5]和 array[9]初始化为 0、2、5、9，则要这样写：

```
int array[10] = {0, 0, 2, 0, 0, 5, 0, 0, 0, 9};
```

上面多余的那些“0”是“被逼”加上去的，当然也可以用其他数字代替。

再来，请定义一个有 100 个 int 的数组，把头 5 个元素赋值为 0、1、2、3、4，把最后 5 个元素初始化为 95、96、97、98、99——噢，再高明的 C 程序员现在也要破口大骂了[1]！

现在，C99 可以让你的初始化随心所欲：

```
int array[10] = {[4] = 2 };
int array[10] = {[0] = 0, [2] = 2, [5] = 5, [9] = 9};
int array[100] = {
    [0] = 0, 1, 2, 3, 4,
    [95] = 95, 96, 97, 98, 99
};
```

这就是 Designated Initializer 的威力，我们可以随便选择某个（些）元素进行初始化，当遇到比较大的数组时，这项特性就显得尤其方便。

对于初始化 struct 和 union，C99 同样有更方便的语法，例如有一个 struct 定义：

```
struct demo
{
    int n;
    double d;
    char array[5];
};
```

现在要求初始化一个 demo 变量，把它的成员 d 赋值为 3.5，把 array[4]赋值为“M”：

C90 要这样写：

```
struct demo s = {0, 3.5, {'\0', '\0', '\0', '\0', 'M'}};
```

或者懒一点：

```
struct demo s = {0, 3.5, 0, 0, 0, 0, 'M'};
```

这些语法要么臃肿要么不直观，C99 允许这样写：

```
struct demo s = {.d = 3.5, .array[4] = 'M'};
```

或者：

```
struct demo s = {.d = 3.5, .array = {[4] = 'M'}};
```

前一种写法是直接把 demo.array[4]当作成员进行初始化，后一种写法是把 demo.array 当作 demo 的子对象（因为 array 是数组）然后去初始化其中的一个元素。

如果要定义一个 demo 数组，内含 10 个元素，把首尾的元素分别初始化为{4, 7.8, "ABC"}和{8, 3.2, "XYZ"}，则可以这样写：

```
struct demo demoArray[10] = {
    [0] = {.n = 4, .d = 7.8, .array = "ABC"},
    [9] = {.n = 8, .d = 3.2, .array = "XYZ"}
};
```

下面我们再讲讲 Compound Literal。

C99 的这项特性主要是为了允许我们在程序中使用（基于数组、结构体等复合类型的）匿名对象。

平时，对于简单数据类型，我们能够直接使用这种类型的常数，例如：

```
void fun(int a);
...
fun(4);
...
```

上面的代码直接用 int 常数“4”作为参数调用 fun()。

要是 fun 的参数类型是一个数组或者结构呢？例如：

```
struct demo
{
    int n;
    double d;
};
...
void fun(struct demo);
```

现在要调用 fun，参数的值是{2, 6.9}，怎么办？

在 C90 中，我们必须先定义一个 demo 变量（哪怕是临时的），然后再压栈：

```
...
struct demo s = {2, 6.9};       //总得起个名字代表{2, 6.9}
fun(s);                         //有了名字的对象就可以压栈了
...
```

而C99把定义临时对象的步骤给省了，因为在这种情况下我们根本不需要：

```
...
fun( (struct demo){2, 6.9} );
...
```

注意写法，(type) {value, ...}表示一个匿名对象，它的类型是type，它的值由{value, ...}给出。

以下是一个关于数组的例子：

```
void fun(int array[2][3]);
...
fun( (int [2][3]) {
                   1, 2, 3,
                   4, 5, 6
                   }
    );
...
```

Compound Literal还可以与前面说的Designated Initializer配合起来：

```
struct demo
{
    int n;
    double d;
};
...
void fun(struct demo s);
void fun2(struct demo *p);
...
```

```
fun( (struct demo){.n = 2, .d = 8.5} );
fun2( &(struct demo){.n = 5, .d = 7.4} );
...
```

上面举的例子都是关于 Compound Literal 在传递函数参数方面的应用，不过，并不是只有在这种情况下才能使用 Compound Literal，实际上，完全可以想用就用：

```
C01   /* Example code 36-01, file name: 3601.c */
C02
C03   struct demo
C04   {
C05   int n;
C06   double d;
C07   };
C08
C09   struct demo *pDemo = &(struct demo){.n = 2, .d = 5.6};
C10
C11   int main(void)
C12   {
C13       pDemo->n++;
C14       int *pi = (int []){0, 1, 2, 3, 4};
C15       for(int i = 0; i < 5; ++i)
C16           pi[i] %= 2;
C17   }
```

```
$gcc -std=c99 3601.c
$
```

在程序运行到C12时，pDemo指向的外部匿名对象的值是：

{2, 5.6}

到执行完C13时，这个对象的值变为：

{3, 5.6}

然后，C14把一个内部匿名数组（此数组有5个int值）的地址赋给pi，到执行完C16时，匿名数组的值变为：

{0, 1, 0, 1, 0}

对于36-01，我们一共使用两个匿名对象，现在我们要讨论一下这些匿名对象的存储性质。

C99明确指出：如果匿名对象出现在函数的外面，则它应该具有静态存储特性（static storage duration），否则它就具有自动存储特性（automatic storage duration）。通俗地说，在函数外面出现的匿名对象（例如上面pDemo指向的匿名对象）应该放在数据段（并且具有内部链接性质），否则，那些内部匿名对象就应该放在栈里面（譬如上面pi指向的匿名对象）。并且，（和普通对象一样，）对于具有自动存储特性的匿名对象，它们的初始化列表中可以包含非常量表达式：[2]

```
void fun()
{
    int a = 4;
    ...
    int *p = (int []){a, a*a, a/2, 123};          //OK in C99
    ...
}
```

“a”、“a*a”、“a/2”都不是常量表达式，但由于匿名对象：

(int []){a, a*a, a/2, 123}

具有自动存储特性，被安排放在栈里面，所以上面代码是正确的。

至于具有静态存储特性的匿名对象，它们被编译器安排在数据段，而对象的初始值必须在程序开始运行之前确定下来，所以这类匿名对象的初始化列表中只能出现常量表

达式。

我们还可以进一步定义“只读（read only)”的 Compound Literal：

```
const int *p = (const int []){0, 1, 2, 3, 4};
```

编译器同样会根据 Compound Literal 出现的地方确定它的存储性质是静态的还是自动的。

最后提一下匿名对象的某些局限性，很明显，由于匿名对象没有名字，所以在某些场合无法“引用自己”。

先来看看普通对象能够做到的事情：

```
struct node
{
    int i;
    struct node *next;
};
...
void fun(struct node);
...
struct node endless = {-1, &endless};    //必须创建一个命名对象
fun(endless);                            //把它作为参数调用 fun
...
```

大家试试能否直接用匿名对象（这个对象的 next 域必须指向自己）调用 fun？[3]

[FN3601]：当然，这种情况可以用 for 语句进行赋值，但这里说的是在定义的同时进行初始化。

[FN3602]：在C90里，结构体等复合类型变量的初始化列表中只能出现常量表达式，不管是外部变量还是内部变量。到了C99，内部的（非静态）复合变量已经可以用非常量表达式去初始化，而C++则允许复合变量用非常量表达式初始化，无论是外部的还是内部的，无论是静态的还是非静态的。

[FN3603]：有兴趣的读者请参阅[Meyers, 2001]和[Meyers, 2001a]。

37 Restricted Pointer

C99 新增加了关键字“restrict”，它可以用来修饰指针类型，例如：

```
int * restrict p;
```

上面表示 int 指针 p 是一个“受限指针（restricted pointer）。

用 restrict 修饰指针主要是为了通知编译器，这个指针指向的内存区域和其他同样用 restrict 修饰的指针指向的内存区域不存在重叠，于是编译器可以根据具体情况决定是否对代码进行优化编译。

举个例子，C 标准库提供两个函数帮助用户把一块内存区域的内容复制到另一块区域，它们分别是 memcpy 和 memmove，下面是它们（在 C90 中）的原型：

```
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
```

这两个函数的作用都是把 n 个字节从 s2 指向的区域复制到 s1 指向的区域，区别是：memcpy 对 s1、s2 有个假定，那就是 s1 和 s2 指向的区域不存在重叠，见图 37-1：

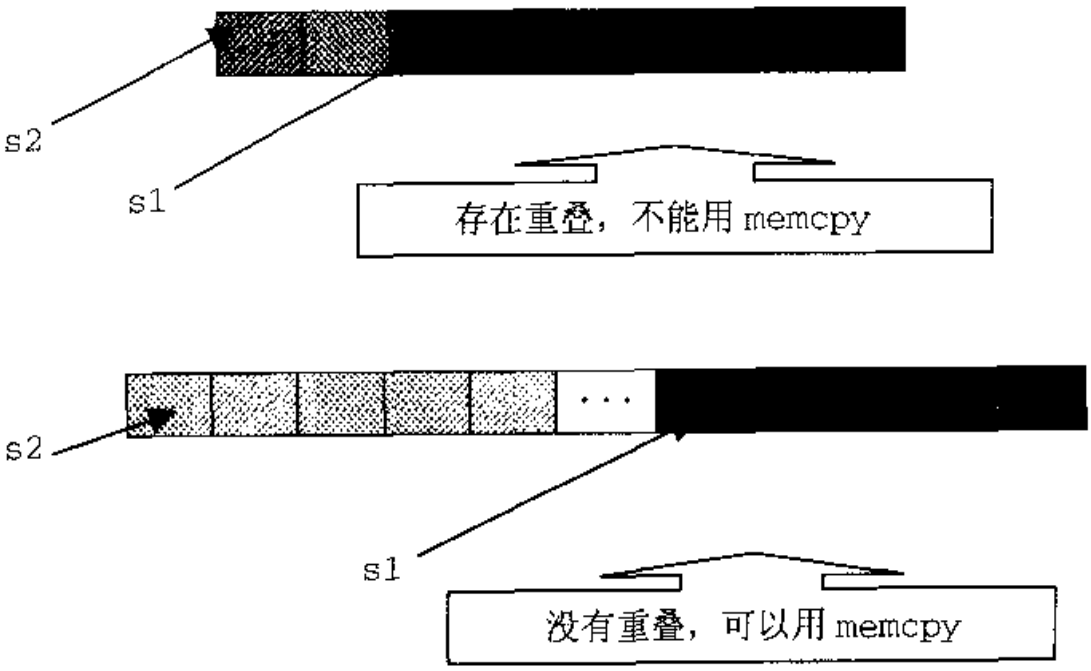


图 37-1

如果我们看一下 memcpy 函数的源代码就更清楚了：

```
C01    void *memcpy(void *s1, const void *s2, size_t n)
C02    {
C03        char *target = s1;
C04        const char *source = s2;
C05        while(n-- > 0)
C06            *(target ++) = *(source ++);
C07        return s1;
C08    }
```

可以看到，memcpy 所做的工作很简单，把 s2 指向的内容逐个字节复制到 s1，递增 s1、s2，直到复制完成。完全能够想象，对相互存在重叠的 s1、s2 使用 memcpy 肯定是不正确的[1]。

在 s1、s2 存在重叠的情况下，我们应该使用 memmove 函数，它能够保证 s2 指向的 n 个字节最后都被正确地复制到 s1 指向的区域。

那这跟本节的内容有什么关系呢？

关键在于某些架构的计算机能够对类似 memcpy 这样的函数进行相当大程度的优化，借助于特殊硬件，这些计算机对两段互不重叠区域的内存复制操作优化后效率有很大的提高。于是，C99 引入了受限指针的概念，这样，这些平台上的 C 编译器就能够根据指针是否使用 restrict 修饰来判断是否对代码进行优化。于是，memcpy 在 C99 的原型就变成：

```
void *memcpy(void * restrict s1,
             const void * restrict s2,
             size_t n);
```

这里要注意两点：

#1 编译器可以优化代码，也可以忽略 restrict，因为并不是所有的计算机平台都可以进行优化，为了保证 C 代码的可移植性，不同平台的 C 编译器有权决定是否根据 restrict 进行代码优化。

#2 程序员一旦使用 restrict 修饰指针，就得自己来保证这些指针指向的区域的的确确不存在重叠，否则会导致严重错误。例如，memcpy 函数本身就是假定不存在重叠区域的，所以它的原型就使用了 restrict 修饰参数，而 memmove 函数的原型就没有使用 restrict，因为它并不保证参数指向的内存不存在重叠区域。

在函数的参数中，数组会被转化为指针进行运算，所以 restrict 还可以出现在数组参数的“[]”里，例如：

在前面我们已经知道：

```
void fun(int array[][5]);
```

和

```
void fun(int (*array)[5]);
```

是等价的。

现在，我们如果要用 restrict 修饰数组指针 array，可以这样写：

```
void fun(int (* restrict array)[5]);
```

同时，C99 允许写成这样：

```
void fun(int array[restrict][5]);
```

这样也可以：

```
void fun(int array[restrict 3][5]);
```

反正那个“3”在这种情况下是多余的。

上面的 3 个原型都是等价的，任何一种形式都被 C99 编译器认可。

最后，说一个有关联的题外话：我们惊奇地发现关键字“static”，一个多才多艺的家伙，又一次扮演新角色，在 C99 中，static 还可以用在函数原型中对数组最高维数进行修饰：

```
void fun(int array[static 3][5]);
```

上面声明了一个函数 fun，它的参数 array 是一个数组指针[2]，它指向有 5 个 int 元素的数组，并且这样的数组至少有 3 个。有了这个信息，编译器将有机会对 fun 作出某种优化。

当然，可以同时使用 restrict 和 static 修饰数组，例如：

```
void fun(int array[restrict static 3][5]);
```

或：

```
void fun(int array[static restrict 3][5]);
```

上面的原型都表示 fun 的参数 array 是一个受限指针，它指向有 5 个 int 元素的数组，并且这样的数组至少有 3 个。[3]

[FN3701]：ISO C 规定，这样的做法会导致不确定的（错误）结果（undefined behavior）。

[FN3702]：别忘了，在函数的参数里面，所有数组都会被编译器自动转换成指针类型。

[FN3703]：有兴趣的读者请参阅[Meyers, 2000a]。

38 增强的数值运算（上）

C99 与 C95 相比，新增加 6 个头文件，它们分别是：

stdbool.h	fenv.h	stdint.h
complex.h	inttypes.h	tgmath.h

除了 stdbool.h 是为了引入布尔类型外，其余 5 个都是关于数值运算的，可见 C 标准委员会大力扩展 C 语言数值运算功能的决心。我们已经讨论过关于复数运算的 complex.h，下面将分析剩下的 4 个头文件。

在分析之前，首先告诉大家，C99 的整数类型多了两种：

```
long long int
unsigned long long int
```

当然，和 long int 可以简写成 long 一样，我们完全可以这样写：

```
long long a = -9223372036854775807LL;
unsigned long long b = 18446744073709551615ULL;
```

于是，C99 的整数、浮点数类型一共有下面这些：

整数	char	unsigned char
	short	unsigned short
	int	unsigned int
	long	unsigned long
	long long	unsigned long long
浮点数	float	
	double	
	long double	

那么，究竟这些类型各占多少个字节呢？下面我们先讨论一下整数类型。

其实，C 标准除了定义 1 个字节是 8 位之外，根本没有硬性规定某个整数类型占多少字节，标准仅仅指出各种整数类型的最小数值范围，并且规定 long long 必须能够表达 long 的所有取值，long 必须能够表达 int 的所有取值，int 必须能够表达 short 的所有取值，详见表 38-1。

表 38-1

	至少能够覆盖到的数值范围	
	最小值	最大值
char	-127 即 $-(2^7-1)$	+127 即 2^7-1
	0	255 即 2^8-1
unsigned char	0	255 即 2^8-1
short	-32767 即 $-(2^{15}-1)$	+32767 即 $2^{15}-1$
unsigned short	0	+65535 即 $2^{16}-1$
int	-32767 即 $-(2^{15}-1)$	+32767 即 $2^{15}-1$
unsigned int	0	+65535 即 $2^{16}-1$
long	-2147483647 即 $-(2^{31}-1)$	+2147483647 即 $2^{31}-1$
unsigned long	0	4294967295 即 $2^{32}-1$
long long	-9223372036854775807 即 $-(2^{63}-1)$	+9223372036854775807 即 $2^{63}-1$
unsigned long long	0	18446744073709551615 即 $2^{64}-1$

各种平台上的编译器会根据实际情况（在符合 C 标准的前提下）决定这些整数类型所占的字节数，譬如说，在我们的 IA-32 平台上，机器字长是 32 位，于是 gcc 的做法是规定：

类型	所占字节数
char	1
short	2

int	4
long	4
long long	8

而在 SPARC64 平台上，机器字长是 64 位，gcc 就这样规定：

类型	所占字节数
char	1
short	2
int	4
long	8
long long	8

读者可以自己检查一下，上面两种做法都是完全符合 C 语言标准的。

要想知道具体某个平台上 C 编译器对这些整数类型所占字节数的规定，可以查看头文件 limit.h ，它是 C 标准指定用来描述各种整数类型的数值范围的标准头文件，例如下面是 IA-32 平台上 gcc 的 limit.h 的一部分：

```
#ifndef _LIMITS_H___
#define _LIMITS_H___

    #define CHAR_BIT 8
    #define SCHAR_MIN (-128)
    #define SCHAR_MAX 127
    #define UCHAR_MAX 255

    #define SHRT_MIN (-32767-1)
    #define SHRT_MIN (-32767-1)
    #define SHRT_MAX 32767
    #define USHRT_MAX 65535

    #ifndef __INT_MAX__
```

```
        #define __INT_MAX__ 2147483647
    #endif
    #define INT_MIN (-INT_MAX-1)
    #define INT_MAX __INT_MAX__
    #define UINT_MAX (INT_MAX * 2U + 1)

    #ifndef __LONG_MAX__
        #define __LONG_MAX__ 2147483647L
    #endif
    #define LONG_MIN (-LONG_MAX-1)
    #define LONG_MAX __LONG_MAX__
    #define ULONG_MAX (LONG_MAX * 2UL + 1)

    #ifndef __LONG_LONG_MAX__
        #define __LONG_LONG_MAX__ 9223372036854775807LL
    #endif
    #define LLONG_MIN (-LLONG_MAX-1)
    #define LLONG_MAX __LONG_LONG_MAX__
    #define ULLONG_MAX (LLONG_MAX * 2ULL + 1)

#endif
```

上面的 SCHAR_MIN、SHRT_MIN、INT_MIN、LONG_MIN 和 LLONG_MIN 分别表示 char、short、int、long 和 long long 类型的最小取值；SCHAR_MAX、SHRT_MAX、INT_MAX、LONG_MAX 和 LLONG_MAX 分别表示 char、short、int、long 和 long long 类型的最大取值；UCHAR_MAX、USHRT_MAX、UINT_MAX、ULONG_MAX 和 ULLONG_MAX 分别表示 unsigned char、unsigned short、unsigned int、unsigned long 和 unsigned long long 类型的最大取值。读者可以自己验证这些定义同样符合 C 语言标准。

由于扩充了整数类型，现在，标准 C 能够处理的整数范围大大扩展，随之而来的是整型常数的类型确定规则及整型变量的类型提升（promotion）规则的调整。

当我们使用整型常数的时候，编译器必须确定常数的类型，例如：

```
int a = 4567;
```

表示把常数 4567 赋给 int 变量 a，但是，这个“4567”本身的类型是什么呢？由于赋值的过程中可能涉及到类型转换，而这些类型转换在不同的平台是不一样的，所以，C99 为确定整型常数的类型提供一个无歧义的统一规定，请见表 38-2。

表 38-2

常数后缀	十进制表示的常数	八或十六进制表示的常数
无	int long long long	int unsigned int long unsigned long long long unsigned long long
u，U	unsigned int unsigned long unsigned long long	unsigned int unsigned long unsigned long long
l，L	long long long	long unsigned long long long unsigned long long
ul，Ul uL，UL	unsigned long unsigned long long	unsigned long unsigned long long
ll，LL	long long	long long unsigned long long
ull，Ull uLL，ULL	unsigned long long	unsigned long long

整型常数的类型由表 38-2 相应项目中从上往下数起第一个能够正确表示它的类型决定。例如（在 IA-32 平台）：

常数	类型
1234	int
-1234**L**	long
1234**U**	unsigned int
1234**UL**	unsigned long
-1234**LL**	long long
1234**ULL**	unsigned long long
0xFFFF8888**U**	unsigned int
-123456789098765**L**	long long [2]
45645645698798 7**U**	unsigned long long

不同类型的整型变量在参与运算时，编译器会暂时提升某些变量的类型，例如：

```
short s = 2;
int i = 3;
int n = s + i;
```

上面的代码在计算 n 时，由于 s 和 i 不属于同一种整数类型，所以编译器必须把 s 从较低级别（rank）的类型（short）暂时提升到较高级别的类型（int），然后再进行运算，最后把结果赋给 int 变量 n。

下面是 C99 整数类型的级别：

类型	
_Bool	↓ 级别越来越高
char	
short	
int	
long	
long long	

从上往下级别是越来越高，并且对于同一类型，带符号和无符号的级别是相同的，即 int 和 unsigned int 级别相等，以此类推。

C99 对整型进行提升的规则如下：

#1 如果两边的操作数类型相同则不需要任何提升。

#2 如果两边都是带符号类型或都是无符号类型，则把低级别类型的操作数提升为高级别的类型。

例如：

```
short a = 2;
long b = 3;
...(a + b)...
```

先把 a（暂时）从 short 提升到 long，然后参与运算，表达式(a + b)得出的结果是 long 类型。

#3 如果两边操作数一个是带符号的（用 S 表示），另一个是无符号的（用 U 表示），并且无符号操作数 U 的类型级别高于或等于带符号操作数 S 的类型级别，则把带符号操作数 S 转换为无符号操作数 U 的类型。

例如：

```
int a = -5;
unsigned long b = 3;
...(a + b)...
```

先把 a（暂时）转换为 unsigned long，然后与 b 相加，(a + b)得到的结果是 unsigned long 类型。值得注意的是，在 IA-32 平台，int 和 long 所占字节数是相同的，这里的 a 原本等于-5，转换成 unsigned long 后，相同的值就有了不同的解释，-5 相当于 0xFFFFFFFC，按 unsigned long 来解释的话，0xFFFFFFFC 是 4294967292。

#4 如果两个操作数，一个是带符号的(用 S 表示)，另一个是无符号的(用 U 表示)，

并且 s 的类型能够表示 u 的类型的所有数值，则把 u 转换为 s 的类型。

例如：

```
unsigned short a = 8;
long b = -9;
...(a + b)...
```

先把 a（暂时）转换为 long，然后参与加法运算，(a + b)的结果属于 long 类型。

#5 如果两个操作数一个是带符号的（用 s 表示），另一个是无符号的（用 u 表示），并且以上规则都不适用，则把 s 和 u 转换为与 s 的类型级别相同的无符号类型。

例如：

```
unsigned int a = 5;
long b = -4;
...(a + b)...
```

由于 b 的类型级别比 a 高，但（在 IA-32 平台上）long 不能表示 unsigned int 的所有数值，所以编译器是把 a、b 都转换为 unsigned long，然后相加，结果自然属于 unsigned long 类型，注意，这种情况下同样存在相同数值转换后解释不一样的问题。

[FN3801]：注意，虽然用了“L”后缀，但最后常数的类型仍然是 long long。

39 增强的数值运算（中）

看了上一节的内容，有些读者可能开始不耐烦了，“这么麻烦啊！同一个 long 整型，在不同平台上还占用不同的字节数？”他们迫不及待地提问：“我怎样才可以指定某个长度的整数类型呢？譬如说，我希望我使用的整型在所有平台上都是带符号的 32 位整数，C 标准能提供统一的方法吗？”

是的，C 标准委员会也考虑到这个问题，并且在 C99 中正式把解决问题的办法标准化。我们的救星在这里：stdint.h。

正如头文件的名称，文件中包含了一系列关于统一的、标准的整数类型的宏定义，通过使用这些宏，我们就能够保证在各种平台上得到统一的整数类型。

stdint.h 定义了很多宏，它们分为几大类：

#1 精确的、指定长度的整数类型定义，主要包括：

int8_t	uint8_t
int16_t	uint16_t
int32_t	uint32_t
int64_t	uint64_t

上面，intN_t 表示 N 位带符号整数类型，uintN_t 表示 N 位无符号整数类型。

我们只要使用这些宏，就可以定义长度（跨平台）固定的整数类型，譬如：

```
#include<stdint.h>
...
uint32_t a;
```

C 预处理器会根据所在平台的具体定义把 uint32_t 替换成（该平台上）合适的整数类型。

我们可以看看 gcc（在 IA-32 平台）的实现：

```
typedef   signed char           int8_t;
typedef   short                 int16_t;
typedef   int                   int32_t;
typedef   long long             int64_t;
typedef   unsigned char         uint8_t;
typedef   unsigned short        uint16_t;
typedef   unsigned int          uint32_t;
typedef   unsigned long long    uint64_t
```

#2 满足指定长度的最低级别整数类型定义：

```
int_least8_t        uint_least8_t
int_least16_t       uint_least16_t
int_least32_t       uint_least32_t
int_least64_t       uint_least64_t
```

上面 int_leastN_t 表示在所有带符号整型中，长度至少为 N 位的最低级别者；uint_leastN_t 表示在所有无符号整型中，长度至少为 N 位的最低级别者。

看看 gcc 的相关定义：

```
typedef   signed char           int_least8_t;
typedef   short                 int_least16_t;
typedef   int                   int_least32_t;
typedef   long long             int_least64_t;
typedef   unsigned char         uint_least8_t;
typedef   unsigned short        uint_least16_t;
typedef   unsigned int          uint_least32_t;
typedef   unsigned long long    uint_least64_t;
```

#3 满足指定长度的最快整数类型定义：

int_fast8_t	uint_fast8_t
int_fast16_t	uint_fast16_t
int_fast32_t	uint_fast32_t
int_fast64_t	uint_fast64_t

上面 int_fastN_t 表示在所有带符号整数中，长度至少是 N 位的运算速度最快的整数类型，uint_fastN_t 表示在所有无符号整数中，长度至少是 N 位的运算速度最快的整数类型。gcc 的实现如下：

```
typedef   signed char           int_fast8_t;
typedef   int                   int_fast16_t;
typedef   int                   int_fast32_t;
typedef   long long             int_fast64_t;
typedef   unsigned char         uint_fast8_t;
typedef   unsigned int          uint_fast16_t;
typedef   unsigned int          uint_fast32_t;
typedef   unsigned long long    uint_fast64_t;
```

#4 在 stdint.h 中定义的各种整数类型的最大、最小值：

INT8_MIN	INT8_MAX	UINT8_MAX
INT16_MIN	INT16_MAX	UINT16_MAX
INT32_MIN	INT32_MAX	UINT32_MAX
INT64_MIN	INT64_MAX	UINT64_MAX
INT_LEAST8_MIN	INT_LEAST8_MAX	UINT_LEAST8_MAX
INT_LEAST16_MIN	INT_LEAST16_MAX	UINT_LEAST16_MAX
INT_LEAST32_MIN	INT_LEAST32_MAX	UINT_LEAST32_MAX
INT_LEAST64_MIN	INT_LEAST64_MAX	UINT_LEAST64_MAX
INT_FAST8_MIN	INT_FAST8_MAX	UINT_FAST8_MAX
INT_FAST16_MIN	INT_FAST16_MAX	UINT_FAST16_MAX
INT_FAST32_MIN	INT_FAST32_MAX	UINT_FAST32_MAX

INT_FAST64_MIN　　INT_FAST64_MAX　　UINT_FAST64_MAX

上面 INTN_MIN 表示 intN_t 类型的最小值，INTN_MAX 表示 intN_t 类型的最大值，UINTN_MAX 表示 uintN_t 类型的最大值，INT_LEASTN_MIN 表示 int_leastN_t 类型的最小值，INT_LEASTN_MAX 表示 int_leastN_t 类型的最大值，UINT_LEASTN_MAX 表示 uint_leastN_t 类型的最大值，INT_FASTN_MIN 表示 int_fastN_t 类型的最小值，INT_FASTN_MAX 表示 int_fastN_t 类型的最大值，UINT_FASTN_MAX 表示 uint_fastN_t 类型的最大值。

C99 规定：

INTN_MIN 等于 $-(2^{N-1})$

INTN_MAX 等于 $2^{N-1} - 1$

UINTN_MAX 等于 $2^{N} - 1$

INT_LEASTN_MIN 的绝对值必须大于或等于$(2^{N-1} - 1)$，符号为负

INT_LEASTN_MAX 的绝对值必须大于或等于$(2^{N-1} - 1)$，符号为正

UINT_LEASTN_MAX 的绝对值必须大于或等于$(2^{N} - 1)$，符号为正

INT_FASTN_MIN 的绝对值必须大于或等于$(2^{N-1} - 1)$，符号为负

INT_FASTN_MAX 的绝对值必须大于或等于$(2^{N-1} - 1)$，符号为正

UINT_FASTN_MAX 的绝对值必须大于或等于$(2^{N} - 1)$，符号为正

#5 为常数添加适当后缀的宏：

INT8_C	UINT8_C
INT16_C	UINT16_C
INT32_C	UINT32_C
INT64_C	UINT64_C
INTMAX_C	UINTMAX_C

看看例子就知道如何使用这些宏：

```
#include<stdio.h>
#include<stdint.h>
...
printf("%llu", UINT64_C(1234));
```

经过预处理后。上面的语句就变成：

```
printf("%llu", 1234ULL);
```

后缀“ULL”被添加到常数“1234”的后面，使常数的类型变为unsigned long long（占8个字节），否则，常数1234的类型就是int（占4个字节），这样，压栈就出现错误，因为前面的“%llu”已经指明printf要打印的数值是属于unsigned long long类型。

C99规定，常数经过INTN_C添加后缀后，其类型必须和相应的int_leastN_t一致；常数经过UINTN_C添加后缀后，类型必须和相应的uint_leastN_t一致。

譬如说，如果在某个系统中，int_least64_t是long，则INT64_C添加的后缀就必须是“L”或“l”，这样的话，常数在加上后缀之后就成为long类型的整数。

下面是这些宏的定义[1]：

```
#define  INT8_C(x)     (int_least8_t)(x)
#define  INT16_C(x)    (int_least16_t)(x)
#define  INT32_C(x)    (int_least32_t)(x)
#define  INT64_C(x)    (int_least64_t)(x)
#define  UINT8_C(x)    (uint_least8_t)(x)
#define  UINT16_C(x)   (uint_least16_t)(x)
#define  UINT32_C(x)   (uint_least32_t)(x)
#define  UINT64_C(x)   (uint_least64_t)(x)
```

到目前为止，我们已经介绍了一大堆标准整数类型，这些宏在使用中还有一个问题需要解决，那就是在printf函数家族中如何使用它们。由于我们不知道这些宏在具体某个平台将被替换成什么类型，所以需要有一种办法提供映射，把这些宏转换到内建整

数类型。

譬如说，我们写：

```
int64_t a = INT64_C(1234);
printf("Here is a 64 bit integer: %□\n", a);
```

上面的□里面应该填什么呢？在int64_t映射为long的平台上，应该写“ld”，而在int64_t映射为long long的平台上，就应该写“lld”。

C99在头文件inttypes.h中定义了提供上述转换功能的宏，为我们在printf函数家族里使用标准整数类型扫除障碍，例如有：

PRIdN	PRIdLEASTN	PRIdFASTN
PRIiN	PRIiLEASTN	PRIiFASTN

上面是部分为printf打印十进制整数提供映射的宏，其中，PRIdN （或PRIiN，两者均可，以下同）对应系统的intN_t类型，PRIdLEASTN 对应int_leastN_t类型，PRIdFASTN 对应int_fastN_t类型。

它们的用法很简单，例如：

```
C01   /* Example code 39-01, file name: 3901.c */
C02
C03   #include<stdint.h>
C04   #include<inttypes.h>
C05   #include<stdio.h>
C06   int main(void)
C07   {
C08   int64_t a = INT64_C(1234);
C09      printf("64 bit integer: %" PRId64 "\n", a);
C10      return 0;
```

```
C11     }
```

在IA-32平台上，GCC对PRId64的定义是：

```
#define PRId64 "lld"
```

所以经过预处理之后，C09就变成：

```
printf("64 bit integer: %" "lld" "\n", a);
```

然后编译器会合并字符串，于是最终变成：

```
printf("64 bit integer: %lld\n", a);
```

```
$gcc  -std=c99  3901.c
$./a.out
64 bit integer: 1234
$
```

[FN3901]：注意，这里给出的是不同于gcc的另一种实现。gcc的做法是仅仅为常数添加适当的后缀，例如：

```
#define  INT32_C(x)  x##L
```

这样，当参数本身不符合标准时，会引起某些混乱，例如：

```
printf(PRIdLEAST32, "\n",
                    PRId32, "\n",
INT32_C(1234567890987654),
1234);
```

上面，程序员因为某些原因给出了不符合标准的参数：1234567890987654，我们来看看将会发生什么。当预处理程序展开宏时，如果按照gcc的做法，语句就变为：

```
printf("%ld\n%d\n", 1234567890987654L, 1234);
```

显然，第1个数值参数已经超出long的数值范围，于是编译器为它进行类型提升，

变为long long类型，这样的话，实际上压栈的是8个字节！但前面的“%ld”却指示函数第1个压栈的数值是一个long（即4个字节），所以，随后printf函数会把1234567890987654的最高4个字节以为是第2个数值参数，从而打印出来的结果令人莫名其妙。

如果使用正文的实现，则由于类型的强制转换，尽管1234567890987654超出范围，但经过强制转换后压栈的仍然是4个字节，错误不会牵连到随后的第2个数值参数。于是程序员就能够集中精力发现错误所在，不会被奇怪的结果所迷惑。

起初，笔者认为gcc的实现有错误。后来，Randy Meyers指出，gcc的做法也是符合C标准的，因为标准规定：宏参数的正确性由程序员自己负责保证。就是说，你要是给出错误参数，编译器作出什么反应都可以。

40 增强的数值运算（下）

和C90一样，C99只有3种浮点数类型：

float

double

long double

而且，和整数类型类似，C语言标准根本没有“float一定要占4字节（即32位）、double肯定是8个字节（即64位）”之类的硬性规定，因为每一种平台支持的浮点数类型有差异，例如Intel的80387浮点处理器支持以下类型的浮点数：

符合IEEE 754标准的single（单精度浮点数，32位）、double（双精度浮点数，64位）和double extended（扩展双精度浮点数，80位）。

而SPARC-V9的浮点处理器则支持以下类型的浮点数：

符合IEEE 754标准的single、double以及厂商自行定义的quadruple（4倍精度浮点数，128位）[1]。

由于众多厂商的架构都在很大程度上遵循IEEE 754标准，所以，尽管C语言标准没有明确指定，但实际上，C语言在各种平台的实现都有以下约定：float对应于IEEE 754中的single，占4个字节；double对应于IEEE 754中的double，占8个字节；至于long double，则各个平台的实现有很大差异，某些平台没有比double更高精度的浮点类型，就把long double定义成和double一样；对于（带浮点单元的）IA-32系列，gcc定义long double占12字节，对应于Intel 80387的double extended[2]；在SPARC-V9平台上，系统定义long double占16字节[3]。

如果想知道某一平台上浮点数类型的定义及数值范围，可以查看float.h头文件，里面定义了系统的浮点数类型的相关特性，例如指数、有效位数、各种浮点类型的最大最小值、舍入方式等。

例如，对于Intel 80387，gcc的float.h是这样的：

```
#ifndef  _FLOAT_H_
    #define  _FLOAT_H_
    #define  FLT_RADIX         2
        #define  FLT_ROUNDS        1
    #define  FLT_MANT_DIG      24
        #define  FLT_DIG           6
        #define  FLT_EPSILON       1.19209290e-07F
        #define  FLT_MIN_EXP       (-125)
        #define  FLT_MIN           1.17549435e-38F
        #define  FLT_MIN_10_EXP    (-37)
    #define  FLT_MAX_EXP       128
        #define  FLT_MAX           3.40282347e+38F
    #define  FLT_MAX_10_EXP    38
    #define  DBL_MANT_DIG      53
        #define  DBL_DIG           15
    #define  DBL_EPSILON       2.2204460492503131e-16
        #define  DBL_MIN_EXP       (-1021)
        #define  DBL_MIN           2.2250738585072014e-308
        #define  DBL_MIN_10_EXP    (-307)
        #define  DBL_MAX_EXP       1024
        #define  DBL_MAX           1.7976931348623157e+308
        #define  DBL_MAX_10_EXP    308
        #define  LDBL_MANT_DIG     64
        #define  LDBL_DIG          18
    #define  LDBL_EPSILON      1.0842021724855044340le-19L
        #define  LDBL_MIN_EXP      (-16381)
        #define  LDBL_MIN
  3.3621031431120935062 6e-4932L
        #define  LDBL_MIN_10_EXP   (-4931)
        #define  LDBL_MAX_EXP      16384
        #define  LDBL_MAX
  1.1897314953572317650 2e+4932L
        #define  LDBL_MAX_10_EXP   4932
        #define  FLT_EVAL_METHOD   2
```

```
    #define  DECIMAL_DIG        21
#endif
```

为了加强程序对浮点运算环境的控制能力，C99增加了头文件“fenv.h”，里面定义了一系列关于访问、控制系统浮点运算方式的宏、函数和#pragma指令：

宏	FE_ALL_EXCEPT　FE_DFL_ENV　FE_DIVBYZERO FE_DOWNWARD　FE_INEXACT　FE_INVALID FE_OVERFLOW　FE_TONEAREST　FE_UNDERFLOW FE_UPWARD
类型定义	fexcept_t　　fenv_t

函数	int feclearexcept(int);
	int fegetexceptflag(fexcept_t*, int);
	int feraiseexcept(int);
	int fesetexceptflag(const fexcept_t*, int);
	int fetestexcept(int);
	int fegetround(void);
	int fesetround(int);
	int fegetenv(fenv_t*);
	int feholdexcept(fenv_t*);
	int fesetenv(const fenv_t*);
	int feupdateenv(const fenv_t*);

#pragma指令	#pragma STDC FENV_ACCESS [ON|OFF]

总之，C99的数值运算环境比C90前进了一大步。不仅如此，C99在其数学函数库中对每一个数学函数都增加了新版本，例如C90的“math.h”中只有一个正弦函数，原型是：

```
double sin(double x);
```

现在，C99的“math.h”有3个正弦函数：

```
float sinf(float x);
double sin(double x);
long double sinl(long double x);
```

其他数学函数都是如此，带“f”后缀的是float版本，带“l”后缀的是long double版本。我们完全可以根据实际参数的类型调用相应版本的数学函数。

更妙的是，C99还可以根据参数的类型自动调用合适的函数，例如：

```
#include<tgmath.h>
int main(void)
{
    float f;                          //f是单精度浮点实数
    double d;                         //d是双精度浮点实数
    long double ld;                   //ld是扩展双精度浮点实数
    float _Complex fc;                //fc是单精度浮点复数
    double _Complex dc;               //dc是双精度浮点复数
    long double _Complex ldc;         //ldc是扩展双精度浮点复数
    f = sin(f);
    d = sin(d);
    ld = sin(ld);
    fc = sin(fc);
    dc = sin(dc);
    ldc = sin(ldc);
}
```

注意，无论是实数类型还是复数类型，无论是float、double还是long double，上面一律都是直接用sin调用数学函数，C99编译器会根据参数的类型帮我们决定最后调用的函数版本。

对应上面的代码，编译器真正调用的函数如下：

```
sinf(f);
sin(d);
sinl(ld);
csinf(fc);
csin(dc);
csinl(dc);
```

C99的这个功能使得程序员无须小心翼翼地留意参数的具体类型然后在函数前面、后面加上正确的前、后缀也能够调用合适的函数，就像上面的例子，只要统一用“sin”调用函数即可。

怎样才能使用这项特性呢？很简单——包含头文件“tgmath.h”就行。

tgmath.h里面定义了一系列的宏，它们的名字没有前、后缀，例如“sin”、“cos”等，我们在代码中调用的“通用型函数”实际上是这些宏。宏经过预处理器展开后，编译器再根据参数类型进一步决定调用真正合适的函数。[4]

[FN4001]：IEEE754 没有定义 4 倍精度浮点数格式，但据说这已经是事实上的标准，请参阅[Kahan, 1996]。

[FN4002]：虽然Intel 80387的long double格式占80位，但gcc为了数据对齐，加入2个填充字节，所以long double类型占12个字节。

[FN4003]：在SPARC平台上，如果使用Solaris操作系统，则描述浮点数格式的float.h由操作系统给出，并不属于编译器的“管辖”范围。

[FN4004]：有兴趣的读者请参阅[Meyers, 2000b]。

41 字符集与字符编码

看看这个别有风味的C程序：

```
C01    /* Example code 41-01, file name: 4101.c */
C02
C03    ??=include<stdio.h>
C04    int main(int argc, char* argv??(??))
C05    <%
C06    if(argc > 1)
C07            printf("Hello, %s!??/a??/n", argv<:1:>);
C08        return 0;
C09    %>
```

编译后执行：

```
$gcc  4101.c
$./a.out   world
Hello, world!
$
```

是的，尽管看起来有点古怪，但上面的代码完全可以被正确编译。

这便引出我们本节要讨论的问题：究竟C编译器“认识”哪些字符？换句话说，在C语言中，使用哪些字符是合法的？

首先搞清楚两个概念：用来编写C源代码的字符所组成的集合称为“源（代码）字符集（source character set)”，程序执行时能够被系统环境正确解释的字符所组成的集合称为“执行字符集（execution character set)”。这两个字符集都包含各自的基本字符集（basic character set）以及数量不定的扩展字符（extended

characters)。它们之间的关系请见图41-1。

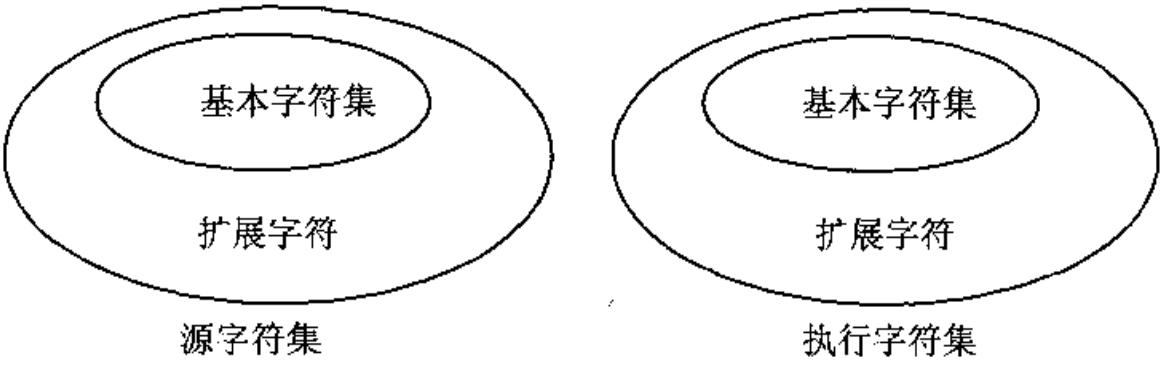


图 41-1

C语言标准规定，无论是基本源字符集还是基本执行字符集，都应该包含以下字符：

26个大写拉丁字母：ABCDEFGHIJKLMNOPQRSTUVWXYZ

26个小写拉丁字母：abcdefghijklmnopqrstuvwxyz

10个阿拉伯数字：0123456789

29个图形字符：! " # % & ' ( ) * + , - . / :
; < = > ? [\ ] ^ _ { | } ~

5个控制字符：空格符（SP）、水平制表符（HT）、垂直制表符（VT）、换页符（FF）、行结束符（end-of-line indicator）

另外，基本执行字符集还应包括以下字符：

null character（一个所有位都清零的字节）

响铃符（BL）、退格符（BS）、回车符（CR）、换行符（NL）

C语言标准还指出：如果在C源文件中除标识符、字符常量、字符串、头文件名、注释和不进行替换的预处理记号外的其他地方出现不属于基本源字符集的字符，将会导致（后果不确定的）错误。

虽然标准没有直接规定上述基本字符集必须遵循ASCII编码，但由于早在1968年，ANSI就批准ASCII编码字符集成为美国国家标准[1]，随后美国的主要计算机厂商基本上都在自己的设备上使用ASCII。所以，ASCII是C语言基本字符集最广泛的编码实现，也就成为事实上的编码标准。

譬如说，根据 ASCII，大写字母“A”的编码是 0x41，编译器读取 C 源文件时会把所有数值是 0x41 的字节认为是字符“A”。现在，假设你的系统使用另一种编码，在这种与众不同的编码中，分号“;”的码值刚好是 0x41，于是，你在自己系统上写的 C 程序里面，凡是出现分号“;”的地方，（遵循 ASCII 的）编译器都会认为是字符“A”。不用说你也知道将会发生什么事——全乱套了。

要想自己的程序能够被世界上尽可能多的编译器成功编译，必须使用 ASCII 来实现 C 的基本源字符集。

后来，ISO 对 ASCII 作出一些很小的改动并在 1983 年批准这个改动后的版本为国际标准[2]，这就是 ISO/IEC 646。ISO 646 规定，各个 ISO 成员国可以在自己的国家标准中把本国额外需要的一些字符安排到原先 ASCII 中“[”、“]”、“{”、“}”等字符占据的位置。例如丹麦需要一些诸如“Å”、“æ”、“ø”的字符表示元音字母，则它可以把这些字符安排进它自己的 ISO 646 实现中去。从而，在丹麦程序员的计算机里很可能没有“[”、“]”、“{”、“}”和“\”这些字符。

于是，问题就出现了。

使用 ASCII 的程序员写：

```
if((argc < 1 )||(*argv[1] == '\0'))
    printf("Hello,%s\n", argv[0]);
```

在丹麦的计算机上却会显示成：

```
if((argc < 1 )øø(*argvÆ1Å) == 'ø0'))
    printf("Hello,%søn", argvÆ0Å);
```

——程序的可读性很差。

鉴于很多采用 ISO 646 的国家都碰到类似的情况，C 标准委员会采取了补救的办法，一些“三联符序列（trigraph sequence）”被指定用来表示那些缺少的字符：

??= 表示 #　　??) 表示]　　??(表示 [

??! 表示 |　　??/ 表示 \　　??- 表示 ~

??> 表示 }　　??< 表示 {　　??' 表示 ^

现在，使用ASCII的程序员和采用ISO 646的程序员之间如果要进行交流，可以使用这种统一、标准的办法：

```
if((argc < 1 )??!??!(*argv??(1??)) == '??/0'))
   printf("Hello,%s??/n", argv??(0??));
```

三联符的加入在某种程度上解决了问题，但C++的设计者们认为这样的方案还不够友好，于是，C++引入一些“替换记号（alternative token)”，为缺乏ASCII字符的C++程序员提供另外一种解决方案：

替换记号	基本记号	替换记号	基本记号	替换记号	基本记号
<%	{	and	&&	and_eq	&=
%>	}	bitor	|	or_eq	|=
<:	[	or	||	xor_eq	^=
:>	]	xor	^	not	!
%:	#	compl	~	not_eq	!=
%:%:	##	bitand	&		

其中，“<%”、“%>”、“<:”、“:>”、“%:”、“%:%:”称为“二联符（digraph)”[3]，其余的都是C++的保留字（reserved word)。

1995年，C语言（C95）正式提供对上述替换记号的支持，不过，支持方式与C++有点差别。对于二联符，两者的做法相同，不需要其他辅助手段，C/C++编译器本身就直接知道二联符的意义；但对于“and”、“bitor”等替换记号，C++把它们看作保留字，所以C++编译器本身就能够“认出”它们，而C95是通过增加头文件“iso646.h”，以宏定义的方式完成记号的替换：

```
#define  and      &&
#define  bitor    |
#define  or       ||
...
```

于是，如果要在C语言中使用“and”、“bitor”这些替换记号，必须包含iso646.h，例如：

```
#include<iso646.h>
int main(void)
{
    int a = 0x12345678 bitor 0x87654321;
    ...
}
```

在C的源文件中，除了基本源字符集里面的字符，我们还可以在以下场合使用部分UCS[4]字符：标识符、字符常量和字符串。

这项特性最先是由C++引入的，随后，C99马上对在源文件中使用UCS字符提供支持。

UCS字符有两种表示方式（n表示任一十六进制数字）：

\unnnn

例如：**\u**4E00，**\u**5F0（相当于\u05F0）

和

\Unnnnnnnn

例如：**\U**00004E00，**\U**5F0（相当于\U000005F0）

\unnnn表示用16位编码表示的UCS字符，\Unnnnnnnn表示用32位编码表示的UCS字符。

在标识符中使用UCS字符很简单，例如：

```
int   \u4E00\u5D9F  =  4;          //定义 int 变量
```

不过，要注意的是，并不是所有 UCS 字符都可以出现在标识符中，C++和 C99 都给出了所有能够在标识符中使用的 UCS 字符集合。

上面讨论了 C 语言的源字符集，下面再讲讲执行字符集。

前面已经提到，基本执行字符集不一定非得使用 ASCII 编码。因为在 C 源代码中，除非直接指定数值，否则基本执行字符集的字符一般都是通过字符常数和字符串的形式进行表示，例如：

```
char c = 'A';                    //char 变量 c 的值是字符“A”
char *pc = "Hello";              //pc 指向字符串“Hello”
printf("%s\n", pc);              //'\n'代表换行符
```

上面三行代码都没有直接用数值去表示字符，这就为编译器进行字符映射提供可能。例如，IBM 的大型机使用 EBCDIC 编码体系，它跟 ASCII 完全不一样。但基于 EBCDIC 的 C 编译器只要把 ASCII 字符映射到 EBCDIC 编码即可解决基本源字符集和基本执行字符集编码不一致的问题。

譬如字符“A”在 EBCDIC 中的码值是 0xC1，于是 IBM 的 C 编译器只须把 0xC1 赋给字符变量 c 即可；又譬如“%”、“s”、换行符“\n”的 EBCDIC 编码分别是 0x6C、0xA2、0x15，那调用 printf 函数之前入栈的指针指向的字符串实际上是：

```
0x6C    0xA2    0x15    0x00
```

总之，基于特殊编码平台的 C 编译器都会提供正确的映射。但是，如果直接写：

```
char c = 0x41;                   //把“A”的 ASCII 值 0x41 直接赋给变量 c
```

就麻烦了。在这种情况下，编译器不可能再提供映射，结果，0x41 在 EBCDIC 编码中对应的是一个绝对不是“A”的字符。

所以，为了提高程序的可移植性，我们最好不要直接为字符变量和字符串指定具体数值。惟一的例外是通过“\u”或“\U”使用 UCS 字符，无论在哪个平台，UCS 字符都会得到相同的解释。

很明显，C 语言的基本执行字符集是单字节字符集，字符数量极其有限，不要说处理像中文这样拥有近十万字符的语种，就算处理一些字符并不太多的语言也显得力不从心。所以，从 1995 年开始，通过增加两个库：wctype.h 和 wchar.h，C 语言提供对宽字符（wide character）和多字节字符（multebyte character）的支持。

在 wchar.h 中，通过 typedef 定义一个宽字符类型：

```
typedef unsigned int wchar_t;
```

下面是 wchar_t 的一些用法：

```
wchar_t wc = 'A';
wchar_t wc = L'A';
wchar_t wc = '\u4E00';
wchar_t wcArray[] = L"ABC";
wchar_t wcArray[] = L"\u4E00\u5F89\u947B";
```

原先针对 char 类型的那些函数都有对应的 wchar_t 版本，例如：

```
<wchar.h>
int fwprintf(FILE * restrict stream,
                  const wchar_t * restrict format, ...);
int fwscanf(FILE * restrict stream,
            const wchar_t * restrict format, ...);
int wprintf(const wchar_t * restrict format, ...);
int wscanf(const wchar_t * restrict format, ...);
...
   <wctype.h>
   int iswalnum(wint_t wc);
int iswalpha(wint_t wc);
int iswcntrl(wint_t wc);
int iswdigit(wint_t wc);
...
```

但是，要注意，C语言标准同样没有指定wchar_t存放的宽字符采用什么编码体系，所以，每个平台都会根据自己的实际情况决定如何去解释wchar_t存放的宽字符。例如，

如果这样写：

```
wchar_t wc = 0x1234;
```

那究竟码值0x1234代表什么字符呢？在这个平台上可能代表某个字符，但在另一个平台上又可能代表其他字符，总之，解释结果是与平台相关的。当然，UCS字符是例外，这前面已经说过了。

最后要说一下，C++同样支持wchar_t，稍有不同的是，wchar_t是C++的关键字。

[FN4101]：请参阅[ANSI, 1986]。

[FN4102]：请参阅[ISO, 1991]。

[FN4103]："%:%:"虽然含有4个字符，但基于习惯，我们还是把它叫做"二联符"。

[FN4104]：UCS指Universal Character Set（统一字符集），它是由国际标准ISO/IEC 10646进行定义，请参阅[ISO, 2000]。

Part V

42 C++的函数

在前面可以看到，一般地，C编译器在生成汇编代码时不改变函数名[1]。

例如定义一个函数：

```
int fun(void){ ... }
```

则汇编代码中标识这段代码的对应符号就是“fun”，调用函数语句对应的汇编代码都是这样的：

```
call fun
```

由于汇编符号仅仅由函数名直接构成，所以在C语言中不可能定义两个名字完全一样的全局函数（即使它们的参数不一样），否则就会因为名字冲突引起编译或链接错误。

C++的情况就复杂得多。

首先，C++允许函数重载（overloading），简单地说，可以同时定义名字相同但参数个数或类型不同的函数，例如：

```
int fun(int n){ ... }
int fun(char c){ ... }
int fun(int n, char c){ ... }
int fun(){ ... }                    //相当于 int fun(void){ ... }
int fun(double, ...){ ... }
```

上面这些函数的名字都是“fun”，但由于参数不同，所以C++允许它们同时存在。当调用函数时，C++编译器会根据给出的参数决定应该调用哪一个版本的函数：

```
fun(4);                  //调用fun(int)
fun('A');                //调用fun(char)
fun();                   //调用fun()
fun(3.67, 8.95)          //调用fun(double, ...)
```

于是，为了实现函数重载这项特性，C++编译器必须有某种办法区分那些名字相同

但参数不同的函数，像C编译器那样仅仅把函数名字作为汇编符号肯定是不行的。很自然，解决问题的方案是通过编码把参数的信息融入到最终输出的汇编符号中，譬如函数：

```
int fun(int){ ... }
```

g++把它的名字“改装”成：

_Z3funi

上面，“_Z”是g++对函数的固定标识，所有函数的最终名字都带有“_Z”；

“3”表示随后的3个字符是一个标识符；

“fun”就是源代码中的函数名；

“i”表示函数带有int类型的参数。

这时，调用函数的汇编代码就变成：

```
push ...                    //参数压栈
call _Z3funi
```

下面再给出一些编码的例子：

函数原型	编码后的函数名
int f();	_Z1fv
int f(int, ...);	_Z1fiz
int f(double);	_Z1fd
int f(char*);	_Z1fPc
int f(const char*);	_Z1fKPc
int f(int [][5]);	_Z1fPA5_i
class ABC; int f(ABC&);	_Z1fR3ABC
class ABC; int f(const ABC*);	_Z1fKP3ABC
typedef void (*PV)(int); int f(PV)	_Z1fPFviE

不仅仅函数重载需要C++编译器为函数名重新编码，面向对象的特性也要求编译器以适当的方式改装函数：

```
class ABC
{
    ...                          //定义数据成员
    public:
    void fun();                  //声明成员函数
    ...
};
```

上面的类ABC有一个成员函数fun，为了区别于普通的全局函数，C++编译器同样需要有效的编码方式为成员函数的名字进行改装，上面的fun函数会被g++扩展编码成：

_ZN3**ABC**3**fun**Ev

可以看出，扩展后的函数名字已经包含了类的名字，这样，在不同的类中我们完全能够定义相同名字并且带相同参数的函数。

由于C++对函数名字的处理，我们如果要调用C的库函数，就必须明确告诉C++编译器，通知它不要扩展函数名：

```
extern "C" int fun(int);          //指出fun是用C编译器编译的函数
...
fun(5);
...
```

上面的函数调用语句对应的汇编代码就变成：

```
push 5
call fun                          //不是 call _Z3funi
```

[FN4201]：准确地说，UNIX世界中的C编译器才是这样的。

43 名字空间

C语言只有一个全局空间，因此，全局性的标识符（例如变量和函数）之间很容易发生名字冲突。如果在开发过程中需要用到两个不同的厂商提供的库，这时，万一这两个厂商对各自的某些函数或变量采用了相同的名字，虽然有办法解决问题，但对于项目管理就增加了难度，费时又费力。所以，在C的年代，开发商一般都使用比较冗长的名字来减少名字冲突的几率。

为了改善这种状况，提供一种逻辑分割的办法把全局空间分解开从而避免大量的名字冲突，C++引入了名字空间（namespace）的概念，例如：

```
C01   /* Example code 43-01, file name: 4301.cpp */
C02
C03   namespace NA {  void fun(){}  }
C04   namespace NB {  void fun(){}  }
C05   namespace NC {  int nVar = 0;  }
C06   int main(void)
C07   {
C08       NC::nVar++;                  //修改NC中的nVar变量
C09       NA::fun();                   //调用NA里的fun函数
C10       using namespace NB;          //明确指定使用NB名字空间
C11       fun();                       //调用的是NB里的fun函数
C12   }
```

注意，上面两个fun函数连参数都是一样的，它们之所以能够共存，不是因为函数重载，而是因为它们分别存在于不同的名字空间里面，从而，C++编译器可以区分它们。

那C++编译器究竟施加了什么魔法，居然让名字、参数完全一样的多个函数和平共

处呢？

其实，到了输出汇编代码的时候，代表这些函数代码的符号一定是不同的。因为汇编语言一般没有什么名字空间，它要求每个符号都是惟一的。所以，C++编译器肯定暗中修改了函数的名字，就像实现函数重载一样，把名字空间的信息也编进最终的符号中。

```
$g++  -S  4301.cpp
```

汇编代码片段如下：

```
A01     .globl _ZN2NC4nVarE
A02         .type  _ZN2NC4nVarE,@object
A03     _ZN2NC4nVarE:
A04         .long 0
A05         ...
A06     .globl _ZN2NA3funEv
A07         .type  _ZN2NA3funEv,@function
A08     _ZN2NA3funEv:
A09         ...
A10     .globl _ZN2NB3funEv
A11         .type  _ZN2NB3funEv,@function
A12     _ZN2NB3funEv:
A13         ...
A14     .globl main
A15     main:
A16         ...
A17         incl   _ZN2NC4nVarE
A18         call   _ZN2NA3funEv
A19         call   _ZN2NB3funEv
A20         ...
```

马上可以看出，_ZN2**NC**4**nVar**E 对应 NC 里面的 int 变量 nVar，_ZN2**NA**3**fun**Ev 对应的是 NA 里面定义的 fun 函数，_ZN2**NB**3**fun**Ev 则对应 NB 里面的那个 fun。毫无疑问，C++编译器给名字空间里面的函数进行了扩展编码，这样的话，只要名字空间不同，即使是名字完全相同的函数或变量也可以多次定义。

另外，如果我们要定义静态外部变量和静态函数，C++的设计者建议使用匿名空间（unamed namespace）而不再用 static。例如：

C 程序员只能依靠 static：

```
static int nVar = 0;
static void fun(void){  }
int main(void)
{
    nVar++;
    fun();
}
```

C 编译器看到 static 修饰变量和函数后，变量或函数的名字就不是全局可见的，于是，其他 C 文件的代码无法访问这些变量和函数。

而 C++程序员应该这样写：

```
C01    /* Example code 43-02, file name: 4302.cpp */
C02
C03    namespace                        //注意，这是--个匿名空间
C04    {
C05        int nVar = 0;
C06        void fun(){}
```

```
C07     }
C08     int main(void)
C09     {
C10         nVar++;
C11         fun();
C12     }
```

C++编译器通过对匿名空间里面的函数、变量进行名字扩展从而使其他文件的代码同样无法访问它们：

```
$g++  -S  4302.cpp
$less  4302.s
```

看一下匿名空间给我们带来什么：

```
A01     .globl _ZN25_GLOBAL__N_4302.cppKrDT9b4nVarE
A02         .type  _ZN25_GLOBAL__N_4302.cppKrDT9b4nVarE,@object
A03     _ZN25_GLOBAL__N_4302.cppKrDT9b4nVarE:
A04         .long  0
A05         ...
A06     .globl _ZN25_GLOBAL__N_4302.cppKrDT9b3funEv
A07         .type
            _ZN25_GLOBAL__N_4302.cppKrDT9b3funEv,@function
A08     _ZN25_GLOBAL__N_4302.cppKrDT9b3funEv:
A09         ...
```

很明显，匿名空间里面的 nVar 和 fun 分别被 g++扩展成：

```
_ZN25_GLOBAL__N_4302.cppKrDT9b4nVarE
_ZN25_GLOBAL__N_4302.cppKrDT9b3funEv
```

虽然有一大串字符，不过，只有 6 个字符值得注意：KrDT9b。这 6 个字符是不断

变化的，如果我们现在重新编译一次 4302.cpp，g++又会给出 6 个新的字符。这样的话就可以确保只有本文件的代码可以访问本文件匿名空间的变量和函数，因为其他文件的代码根本不可能准确知道这 6 个由 g++给出的随机字符。

44 C 和 C++的标准库

本小节我们概览一下C/C++标准库的全貌。

C/C++标准库的内容主要包括：

宏定义　　（譬如：#define FLT_RADIX 2）

类型定义　　（譬如：typedef unsigned wchar_t ）

变量声明和定义　　（譬如：extern int errno; ）

库函数声明和定义　　（譬如：size_t strlen(const char *s); ）

其中，宏定义、类型定义、变量声明和库函数的声明都在标准头文件进行描述，所以，标准头文件实际上已经勾画出整个标准库的轮廓。

现在我们来看看C语言的标准头文件。

C90有这些：

assert.h	ctype.h	errno.h	float.h
limits.h	locale.h	math.h	setjmp.h
signal.h	stdarg.h	stddef.h	stdio.h
stdlib.h	string.h	time.h	

C95增加这几个：

iso646.h	wchar.h	wctype.h

C99再增加下面这些：

complex.h	fenv.h	inttypes.h	stdbool.h
stdint.h	tgmath.h		

要注意的是，高版本的C标准不一定是仅仅增加了新的头文件，还可能对原先低版本C标准的某些头文件进行了修改和扩充。

使用C标准库很简单，只需包含相应的头文件就可以了，例如：

```
/* C代码:  */
#include<stdio.h>
int main
{
    printf("printf() again...boring!\n");
 }
```

而对于C++，情况就有点复杂。

首先，C++的设计者希望C++能在相当大的程度上兼容C，至少，C++编译器应该最大限度地接受已经存在的大量C代码。所以，允许使用C标准库自然是追求的目标之一。但C语言并没有扩展名字这回事，也就是说，如果想直接使用已经用C编译器编译好的C库（主要是指库函数），C++编译器必须知道哪些是C库函数，从而不对其进行名字扩展。于是，链接修饰符 extern "C" 被用来指出哪些函数具有“C语言链接性质”，所以，编译器厂商一般都会在原来的C语言标准头文件增加一些东西：

```
#ifdef __cplusplus
    extern "C" {
#endif
    int xxx();                          //C标准库函数的声明
    int yyy();                          //同上
    ...
#ifdef __cplusplus
    }
#endif
```

根据是否定义宏“__cplusplus”决定是否添加“extern "C" ”修饰符，如果宏被定义，则表明是C++编译器在处理代码，所以必须加上extern "C"；否则就不用添加，因为C编译器不认识extern "C"。

对C标准头文件作了以上处理后，我们就可以在C++代码中调用C库函数：

```
//C++代码:
#include<stdio.h>
int main(void)
{
    printf("calling printf in a cpp file\n");
}
```

至于C++的库（包括宏、类型和函数），直到C++标准出版前一直都是放在全局空间，这些C++头文件和普通的C头文件一样，均带有“.h”后缀，例如“iostream.h”。使用上和C语言没什么不同：

```
//早期的C++代码:
#include<iostream.h>
int main(void)
{
    cout << "using an old style head file" << endl;
}
```

自然，C++编译器会对C++库函数进行名字扩展。

不过，C++标准出来后，情形就不一样了。标准规定，所有C/C++标准库都必须在名字空间“std”里进行定义，也就是说，标准保留出一个特定的名字空间，专门放置C/C++标准库，任何人都不应该在std定义自己的库。

在std里定义的C库所对应的头文件都没有“.h”后缀，并且在前面多了一个字符“c”，下面是C++标准规定的C库头文件[1]：

cassert	cctype	cerrno	cfloat
climits	clocale	cmath	csetjmp
csignal	cstdarg	cstddef	cstdio
cstdlib	cstring	ctime	ciso646
cwchar	cwctype		

现在，使用“C++标准版”的C库，必须明确声明使用std名字空间：

```
//标准的C++代码：
#include<cstdio>
using namespace std;
int main(void)
{
    printf("Don't say me boring please.\n");
}
```

或者：

```
//标准的C++代码：
#include<cstdio>
int main(void)
{
    std::printf("Don't say me boring please.\n");
}
```

同样，在std里定义的C++库对应的头文件也没有“.h”后缀，以下是全体成员：

algorithm	bitset	complex	deque
exception	fstream	functional	iomanip
ios	iosfwd	iostream	istream
iterator	limits	list	locale
map	memory	new	numeric
ostream	queue	set	sstream
stack	stdexcept	streambuf	string
typeinfo	utility	valarray	vector

同样，使用这些C++头文件必须声明使用std名字空间：

```
//标准的C++代码：
#include<iostream>
```

```
using namespace std;
int main(void)
{
    cout << "Yes, you're right...I am boring. " << endl;
}
```

或者：

```
//标准的C++代码：
#include<iostream>
int main(void)
{
    std::cout << "Last time, I swear! " << endl;
}
```

最后，肯定有一些喜欢追问到底的读者会问：那些在名字空间std里面定义的C库函数究竟有没有进行名字扩展呢？

C++标准规定厂商可以自行决定是否扩展。就是说，C标准库既可以这样声明：

```
namespace std
{
    extern "C" int xxx();       //声明C库函数xxx()不进行名字扩展
    extern "C" int yyy();       //同上
    ...
}
```

也可以这样声明：

```
namespace std
{
    extern int xxx();           //声明C库函数xxx()进行名字扩展
    extern int yyy();           //同上
    ...
}
```

前者意味着可以直接使用已有的C库，后者则必须用C++编译器重新编译C库，g++采用了前者。

[FN4401]：注意，这里只有18个头文件，因为C99比C++标准晚推出，所以C++的C库只包含了C95的18个头文件。

45 模板

C++支持模板（template），我们通过模板可以实现一些高级特性，例如“泛型编程（generic programming）”，STL就是这方面的一个经典范例。本节不打算讨论使用模板的技巧，而仅仅以函数模板为例，关注一下模板的底层实现。

以往在没有模板的日子，如果我们想实现一个加法函数，它能够对两个参数求和，并根据参数的类型返回具有适当类型的值，就必须手工书写所有的代码：

```
char sum(char a, char b){   return (a + b);  }
short sum(short a, short b){   return (a + b);  }
int sum(int a, int b){   return (a + b);  }
float sum(float a, float b){   return (a + b);  }
double sum(double a, double b){   return (a + b);  }
...
```

——非常麻烦。C++允许我们用模板表达“通用型函数”：

```
template<typename T>
T sum(T a, T b)
{
    return (a + b);
}
```

现在，C++编译器可以根据我们调用sum函数的参数类型“现场”生成一个适当的函数，然后调用它，例如：

```
int main(void)
{
    float fa, fb, fs;
    fs = sum(fa, fb);
}
```

看到上面的代码，C++编译器就为我们生成一个“float版本”的sum函数并调用

它，所有这一切都不需要我们去干预，而且，如果我们给出的参数类型不一致，则编译器会报错，例如：

```
int na,ns;
float fb;
ns = sum(na, fb);                    //Error!
```

na是int类型，fb是float类型，由于函数模板sum并不支持对两个不同类型的参数求和，所以C++编译器会报告无法生成真正的函数，从而让程序员有机会知道调用参数出了问题。[1]

函数模板并不是真正的函数，它只是C++编译器生成具体函数的一个“模子”，所以，我们不能把函数模板的声明和定义分开放在不同的文件里。譬如，我们在一个头文件声明函数模板，而在另一个C++文件中写出函数模板的定义，然后在自己的源代码文件中包含头文件：

4501.h

```
template<typename T>
T sum(T a, T b);
```

4501.cpp

```
template<typename T>
T sum(T a, T b)
{
    return (a + b);
}
```

4501a.cpp

```
#include"4501.h"
int main(void)
{
    int a, b, s;
    s = sum(a, b);
}
```

这样做可以吗？

不行。问题出在4501.cpp，它里面只是一个函数模板，当编译4501.cpp时，由于编译器没有看到任何调用函数sum的语句，所以并不会生成具体的、真正的函数，导致最后链接程序因为找不到代表具体函数的符号而链接失败。

正确的做法是把整个函数模板的定义放在头文件，然后在需要调用函数的C++代码文件中包含头文件：

4502.h

```
template<typename T>
T sum(T a, T b)
{
    return (a + b);
}
```

4502.cpp

```
#include"4502.h"
int main(void)
{
    int a, b, s;
    s = sum(a, b);
}
```

和其他函数一样，C++编译器在具现函数模板时同样会对生成的函数进行名字扩展：

```
$g++  -S  4502.cpp
$less  4502.s
```

```
A01     .globl main
A02         .type  main,@function
A03     main:
A04         ...
A05         call   _Z3sumIiET_S0_S0_
A06         ...
A07
A08         .section  .gnu.linkonce.t._Z3sumIiET_S0_S0_,
            "ax",@progbits
A09         .weak  _Z3sumIiET_S0_S0_
A10         .type  _Z3sumIiET_S0_S0_,@function
A11     _Z3sumIiET_S0_S0_:
A12         ...
```

可以看出，g++具现的函数经过扩展编码后的名字是：

```
_Z3sumIiET_S0_S0_
```

A05明确指出调用这个函数。

你可能会担心，把函数模板的定义放在头文件，然后每一个需要调用函数的C++源文件都包含它，这不会导致函数被多次定义造成名字冲突吗？例如：

4502.h

```
template<typename T>
T sum(T a, T b)
{
    return (a + b);
}
```

4503.cpp

```
#include"4502.h"
void fun();
int main(void)
{
    int a, b, s;
    s = sum(a, b);
    fun();
}
```

4503a.cpp

```
#include"4502.h"
void fun()
{
    int x, y, z;
    z = sum(x, y);
}
```

```
$g++  4503.cpp  4503a.cpp
```

——会出现链接错误吗？

在main里面调用sum，编译器就具现出函数，它的名字是：

_Z3sumIiET_S0_S0_

在fun里调用sum，编译器同样具现出函数，它的名字也是：

_Z3sumIiET_S0_S0_

于是，最后在全局空间里就存在两个“_Z3sumIiET_S0_S0_”！

——名字冲突的问题又出现？

为了解决这个问题，g++采用了和处理内联函数相同的办法。首先，读者有没有留意到汇编代码中并没有如下语句：

```
.globl _Z3sumIiET_S0_S0_
```

取而代之的是A08、A09，这两行语句告诉编译器，对符号“_Z3sumIiET_S0_S0_”执行“特殊政策”，链接程序只须对任何一个“_Z3sumIiET_S0_S0_”进行链接即可，而不用理会存在多份副本的事实。用nm查看一下目标代码文件：

```
$g++  -c 4503.cpp
$nm  4503.o
00000000 U _Z3funv
0000000c W _Z3sumIiET_S0_S0_
0000002c T main
```

注意到符号_Z3sumIiET_S0_S0_吗？和普通的全局函数不同，它的属性是“W”（weak），而普通全局函数（例如main）的属性是“T”。由于g++作了如此区分，我们便不用担心名字冲突的问题。

不过，g++的处理办法也不是绝对的“无懈可击”。如果函数模板的定义在多个文件中不一致，则最后的结果会取决于链接的次序：

4504.h

```
template<typename T>
T sum(T a, T b)
{
    return (a + b);
}
```

4504.cpp

```
#include"4504.h"
#include<iostream>
void fun();
int main(void)
{
    int a = 1, b = 2;
    std::cout << sum(a, b) << endl;
    fun();
}
```

4504a.h

```
template<typename T>
T sum(T a, T b)
{
    return (a - b);
}
```

4504a.cpp

```
#include"4504a.h"
#include<iostream>
void fun()
{
    int a = 1, b = 2;
    std::cout << sum(a, b) << endl;
}
```

如果：

```
$g++  4504.cpp  4504a.cpp
```

则：

```
$./a.out
3
3
$
```

如果：

```
$g++  4504a.cpp  4504.cpp
```

则：

```
$./a.out
-1
-1
$
```

和处理版本不一致的内联函数一样，C++标准并没有要求编译器对这类问题进行检测。所以，如何保证相同名字的函数模板只有惟一一个版本就只能靠程序员自己。

[FN4501]：如果不是用函数模板而是用普通函数，则即使参数类型不完全一致也可能会通过编译，例如：

```
int fun(double a, double b);
...
int n;
double d;
fun(n, d);                          //OK
```

因为C/C++中，int类型可以自动转换成double类型，于是C++编译器在这种情况下不会报错。

46 外部对象的初始化

在C语言（C90和C99）中，我们必须用常量表达式去初始化外部变量，例如：

```
C01    /* Example code 46-01, file name: 4601.c */
C02
C03    #include<stdio.h>
C04    int a = 7;
C05    static int b = 6 * 3;
C06    int f(void)
C07    {
C08        static int n = 4;
C09        return ++n;
C10    }
C11    Int main(void)
C12    {
C13        printf("Call f() the first time: n = %d\n", f());
C14        printf("Call f() the second time: n = %d\n", f());
C15    }
```

C04定义外部变量a，C05定义静态外部变量b，C08定义静态内部变量n，并分别用常量表达式进行初始化。

看一下C编译器如何处理这些变量的初始化：

```
$gcc  -S  4601.c
$less  4601.s
```

```
A01    .globl a
A02        .data
```

```
A03         .type a,@object
A04         .size a,4
A05     a:
A06         .long 7
A07
A08         .type b,@object
A09         .size b,4
A10     b:
A11         .long 18
A12
A13         .type n.0,@object
A14         .size n.0,4
A15     n.0:
A16         .long 4
A17
A18     .globl f
A19         .type f,@function
A20     f:
A21         ...
A22
A23     .globl main
A24         type main,@function
A25     main:
A26         ...
```

A01~A06是关于变量a的，A08~A11是关于b的，A13~A16是关于n的。

由于a是全局变量，所以A01指示符号a是全局可见的；b和n分别是静态外部变量和静态内部变量，所以没有类似A01的语句，从而它们是局部可见的符号。但无论是a、b或者n，这些变量都由编译器把初始化值直接放在对应的存储空间（A06、A11、A16），所以实际上在程序运行前它们已经完成初始化。

把4601.c改动一下，例如：

```
C01   /* Example code 46-02, file name: 4602.cpp */
C02
C03   #include<stdio.h>
C04   int g(void){   return 9;  }
C05   int a = g() + 4;
C06   static int b = a * (g() + 6);
C07   int f(void)
C08   {
C09       Static int n = a * g() - b;
C10       Return ++n;
C11   }
C12
C13   int main(void)
C14   {
C15       Printf("Call f() the first time: n = %d\n", f());
C16       Printf("Call f() the second time: n = %d\n", f());
C17   }
```

注意一下C05、C06和C09，由于全局变量a、静态外部变量b和静态内部变量n不再用常量表达式进行初始化，所以，C编译器无法编译4602.cpp。

大家猜猜：C++能否接受非常量表达式初始化外部变量？

当初，B.Stroustrup在设计C++的时候，他为C++确立了一个目标，那就是让用户自定义类型（例如各种struct、class等）具有与内建类型（例如int、double等）同样的可用性。既然C语言允许定义、初始化全局的内建类型变量，那C++也应该允许定义、初始化全局（用户定义类型）对象，例如：

```
class A
{
        int m;
    public:
        A(int i = 0);
};
A::A(int i): m(i){  }
```

上面是一个很简单的类定义，根据B.Stroustrup的设计原则，C++应该接受这样的代码，允许我们定义外部对象：

```
A a(5), b;                          //定义并初始化外部对象a和b
int main(void){ }
```

由于初始化对象一般必须调用类的构造函数，所以，对于C++编译器来说，接受上面的代码就意味着必须允许在进入main函数前调用某些用户定义的函数（例如类的构造函数），这些函数的代码实际上比main函数更早运行。

很自然地，满足上面的特性所带来的一个扩展就是C++允许在初始化外部对象时调用任何函数或计算表达式：

```
int f(void);
int g(void);
A a( f()+g() );                     //用f()、g()的返回值之和初始化a
int n = f()*g();                    //用f()、g()的返回值之积初始化n
int main(void){  }
```

因此，C++程序员可以放心地使用非常量表达式初始化外部变量（对象），前面的4602.cpp是完全合法的C++代码。

现在，我们先用一个简单的例子分析一下g++初始化外部对象的实现方法：

```
C01    /* Example code 46-03, file name: 4603.cpp */
C02
C03    int f(void){  return 4;  }
C04    int a = f();
C05    int main(void){  }
```

C04用f()的返回值初始化全局int变量a，虽然是一行很简单的代码，但已经有足够的代表性：

```
$g++  -S  4603.cpp
$less  4603.s
```

```
A01 .globl _Z1fv
A02     .type  _Z1fv,@function
A03 _Z1fv:
A04     ...
A05
A06 .globl main
A07     .type  main,@function
A08 main:
A09     ...
A10
A11 .globl a
A12     .bss
A13     .type  a,@object
A14     .size  a,4
A15 a:
A16     .zero  4
A17
```

```
A18     .type _Z41__static_initialization_and_destruction_0ii,
        @function
A19 _Z41__static_initialization_and_destruction_0ii:
A20     pushl  %ebp
A21     movl   %esp, %ebp
A22     Subl   $8, %esp
A23     cmpl   $65535, 12(%ebp)
A24     jne .L7
A25     cmpl   $1, 8(%ebp)
A26     jne .L7
A27     call   _Z1fv
A28     movl   %eax, a
A29 .L7:
A30     leave
A31     ret
A32
A33     .type  _GLOBAL__I__Z1fv,@function
A34 _GLOBAL__I__Z1fv:
A35     pushl  %ebp
A36     movl   %esp, %ebp
A37     subl   $8, %esp
A38     subl   $8, %esp
A39     pushl  $65535
A40     pushl  $1
A41     call
        _Z41__static_initialization_and_destruction_0ii
A42     addl   $16, %esp
A43     leave
A44     ret
```

A11~A16 是关于符号 a 的，首先，编译器把 a 安排在 BSS 段[1]，初始值是 0。然

后，我们观察到，比起平时上面多出了两段代码（A20~A32、A36~A46），分别用局部符号：

_Z41__static_initialization_and_destruction_0ii

和

_GLOBAL__I__Z1fv

进行标识（A18~19、A34~A35）。

经过分析，这两段代码的调用关系如图46-1所示。

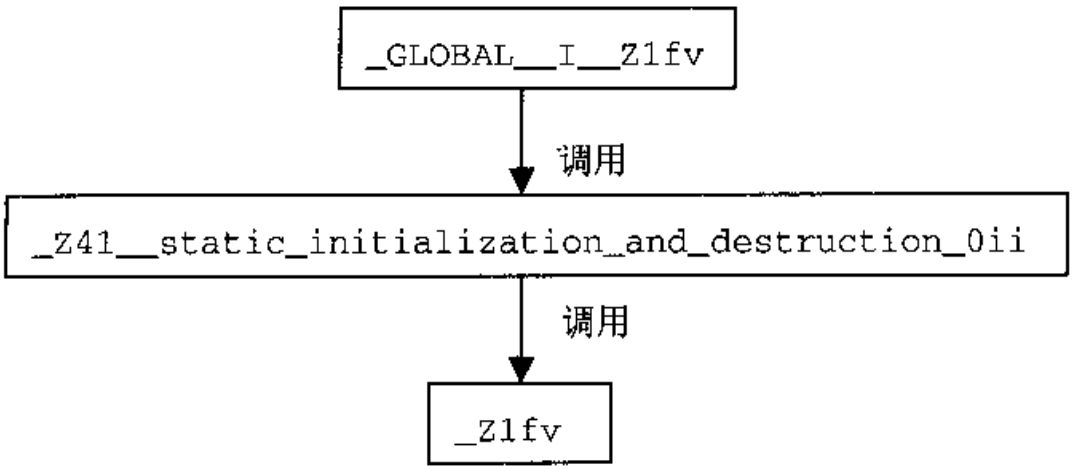


图 46-1

_GLOBAL__I__Z1fv 根据实际情况（构造对象还是析构对象）压入不同的参数（A40~A41），然后调用：

_Z41__static_initialization_and_destruction_0ii

由它根据参数调用_Z1fv（A23~A27），再把_Z1fv 的返回值赋给 a，从而完成 a 的初始化工作。

不同于内部类型变量，用户定义类型的对象不仅有初始化，还有析构的问题，因为对象的析构函数必须在对象生存期结束时被调用。例如：

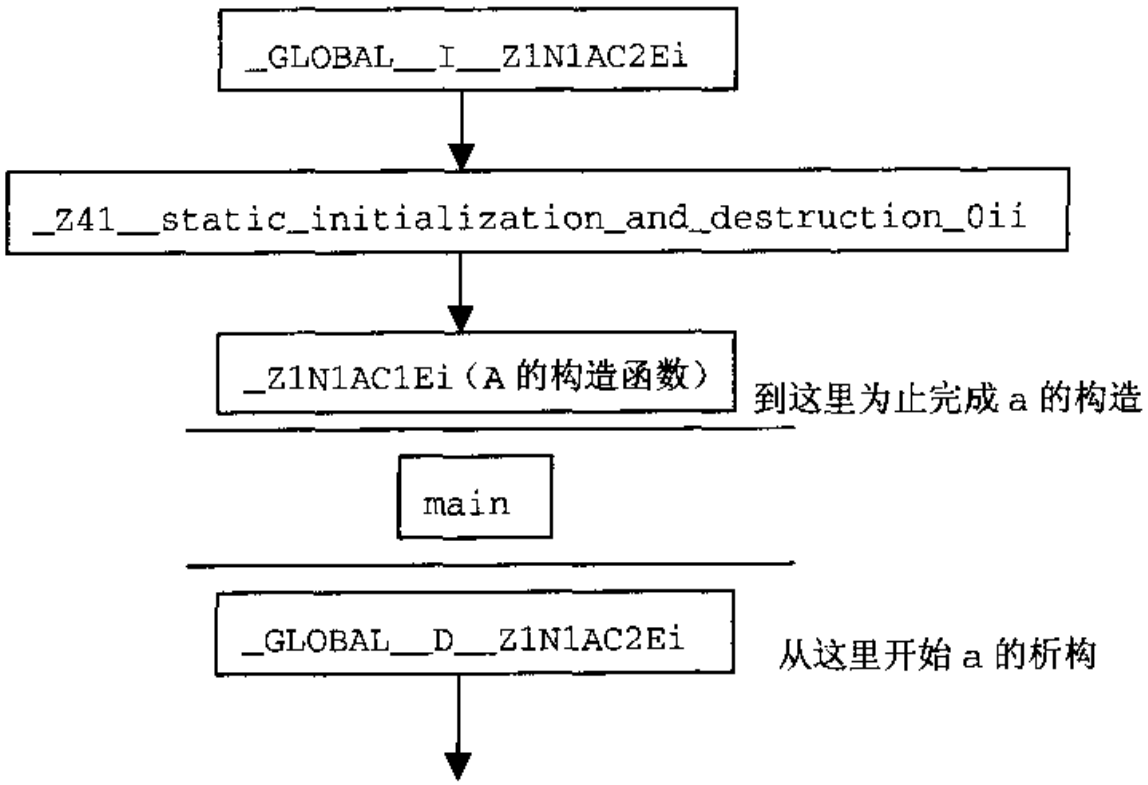
```
C01    /* Example code 46-04, file name: 4604.cpp */
C02
C03    class A
```

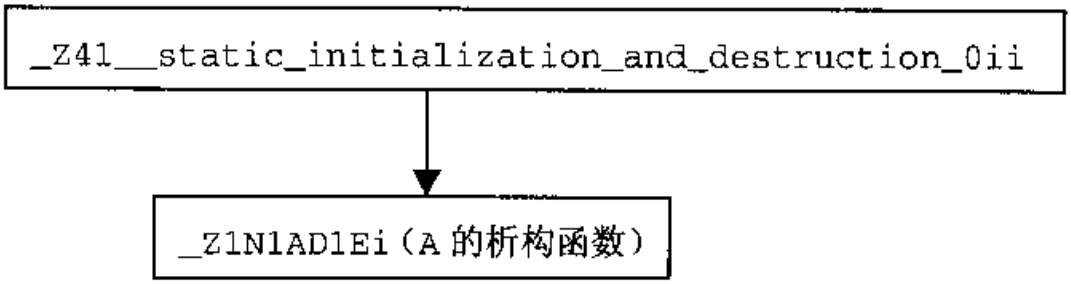
```
C04    {
C05        int m;
C06    public:
C07        A(int i = 0);
C08        ~A();
C09    };
C10    A::A(int i): m(i){  }
C11    A::~A(){   m = 0;  }
C12
C13    A a(5);
C14    int main(void){  }
```

C10 定义 A 的构造函数，C11 定义 A 的析构函数。在 main 函数执行之前，C13 导致 A 的构造函数被调用，而在 main 函数结束后，A 的析构函数必须被执行。

```
$g++  -S  4604.cpp
$less  4604.s
```

分析汇编代码后[2]可以看出全局对象 a 的构造、析构过程如下：





最后，大家必须知道的是，同一个源文件定义的全局对象，C++编译器保证按照它们的定义顺序进行初始化（即构造函数会依照定义顺序被调用），而在 main 函数执行完后，C++编译器同样保证这些全局对象会按照定义顺序的逆序进行析构（即析构函数会根据定义顺序的逆序被调用）。但是，C++标准对不同文件定义的全局对象的构造和析构顺序不进行任何特定的承诺，各个 C++编译器可以有自己的实现，例如：

4605.h

```
class A
{
        int m;
    public:
        A(int i = 0);
        ~A();
};
```

4605.cpp

```
#include"4605.h"
A::A(int i): m(i){  }

A::~A()
{
    m = 0;
}
```

4605a.cpp

```
#include"4605.h"
A a(5);
int main(void){ }
```

4605b.cpp

```
#include"4605.h"
A b;
```

上面分别在 4605a.cpp 和 4605b.cpp 中定义了全局对象 a 和 b，但 a 不一定就先于 b 进行构造，反之亦然；同时，a 也不一定就先于 b 进行析构，反之亦然。

在这种情形下，我们惟一可以保证的是：

在 main 函数执行之前，a 和 b 的构造函数会被调用；在 main 函数执行之后，a

和 b 的析构函数会被调用，仅此而已。

[FN4601]：BSS 段一般用来放置需要程序运行后再初始化的数据。

[FN4602]：限于篇幅，这里不再列出汇编代码。